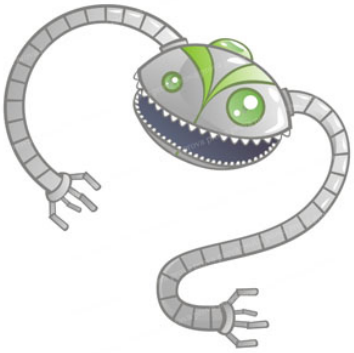
Aprendendo Django no Planeta Terra - vol. 1



Marinho Brandão

1ª edição - 2009

revisão

Mychell Neiva Rodrigues

ilustrações e capa

João Matheus Mazzia de Oliveira

impressão e acabamento

Lulu.com

E-mail: marinho@aprendendodjango.com

www.aprendendodjango.com

Licença

Sumário

Volume 1

Volume 2

Volume 1

Agradecimentos

Há diversas pessoas a agradecer, mas os nomes mais fortes que tenho pra citar são estes:

Mãezinha e Paizinho

Letícia e Tarsila

Miltinho e Verinha

Mychell e João Matheus

Andrews Medina, Guilherme Semente e Betty Vogel

Douglas Adams, Felippe e Donaldson Nardi

Os desenvolvedores do Django e do Python

Linus Torvalds e Richard Stallman

As pessoas acima foram decisivas para que esta obra se tornasse real. Todo agradecimento e dedicatória são poucos diante do que recebi deles.

Dicas de Aprendizado

Antes de entrar de cabeça na leitura do livro, é bom manter em mente alguns princípios que vão facilitar a compreensão, manter a calma em alguns momentos e fortalecer o aprendizado.

Veja abaixo:

Não copie e cole, digite

A pior coisa a fazer para se aprender algo é buscar o atalho de copiar e colar, com aquela velha auto-ilusão "ahh, já entendi isso, vou copiar e colar só pra confirmar que eu entendi".

Isso não *cola* - ou pelo menos não cola na sua memória. Digite cada comando, cada linha, cada coisinha, faça por você mesmo, e os resultados serão mais sólidos e duradouros.

Calma, uma coisa de cada vez

O detalhamento inicial é feito assim por sua própria natureza. À medida que algumas coisas são explicadas detalhadamente, elas passam a ser apontadas de forma mais objetiva dali em diante, e coisas em foco naquele capítulo passam a ser mais detalhadas.

Assim, se você continua pensando que o livro está detalhando demais, é um bom sinal de que aprendeu as lições dos capítulos anteriores de tal forma que nem notou. Parabéns!

Não existe mágica, não se iluda com isso

O Django não é uma ferramenta mágica. Muito menos de truques.

A produtividade do Django está ligada a três coisas muito importantes:

- **Framework.** Trabalhar com um framework é mais produtivo simplesmente porque você não faz coisas que já estão prontas e levaram

anos para serem criadas de forma que você possa usar facilmente agora. Esse valor só tem essa força e dinâmica porque o Django é **software livre**.

- **Conhecimento.** O seu conhecimento do Django e de conceitos aprofundados de **Web**, **Programação Orientada a Objetos** e **Bancos de Dados** faz TODA a diferença.

- **Python.** É uma linguagem de fácil compreensão, não burocrática e prática. Muito prática. Isso evita que seu cérebro se perca em interpretar coisas e símbolos que não precisaria interpretar e mantenha o foco na solução.

Portanto, não existem atalhos, nem vida fácil, mas você pode facilitar as coisas e se divertir com isso, usando as ferramentas certas, e agindo com foco e persistência. Então, numa bela manhã vai perceber que está produzindo com qualidade e rapidez, por seus próprios méritos.

Versão do Django

A versão do Django adotada nesta série é a **versão 1.0.2**.

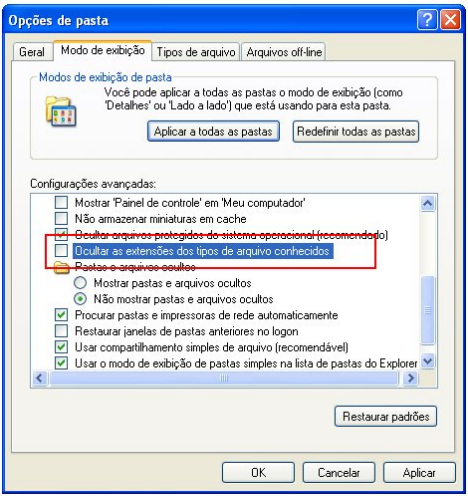
Algumas pessoas têm encontrado algumas dificuldades em decorrência de usarem uma versão diferente. Há diferenças entre versões em aspectos que temos trabalhado, portanto, use a versão **1.0.2 ou superior**.

Fique ligado nas extensões do arquivos

No **Windows**, o padrão é ocultar as extensões dos arquivos. Por exemplo: para um arquivo **"teste.txt"**, será exibido apenas como **"teste"**.

Isso é tremendamente chato para o desenvolvedor.

Para facilitar a sua vida, vá numa janela do **Explorer**, ao menu **Ferramentas** -> **Opções de Pasta** e desabilite a opção que você ve na imagem abaixo.



Apresentação

Olá,

nos últimos anos, tenho investido a maior parte do meu tempo com o **Django**.

Eu vinha de uma longa jornada onde segui firmemente (e nem sempre fielmente ou alegremente) com o Delphi e o PHP. Mas chegou uma hora que uma **nova tecnologia** era necessária - já haviam se passado quase 10 anos!

A saída foi conhecer as linguagens de programação populares, colocando alguns princípios em mente:

- A tecnologia teria que ser **multi-plataforma**
- A tecnologia teria de ser útil para **sites e sistemas web**
- Eu queria ter **resultados**

Após meses de namoro com Java e .Net, notei que aquilo não era o que eu precisava. Então parti para conhecer outras... TCL, Ruby, **Python**... opa! Não precisei passar daqui!

Do Python eu cheguei ao Django e ali fiquei.

O Django é uma daquelas coisas que você já tem uma boa impressão na chegada, mas algumas semanas depois, você está completamente apaixonado. Hoje, quase 3 anos depois, **eu sinto amor pra vida inteira** - ou pelo menos para a curta vida que as tecnologias costumam levar.

Há cerca de um ano atrás, estava para concluir a escrita de um livro em português sobre Django, e percebi que ele não era nada do que eu queria.

Passei a borracha e resolvi esperar.

Então depois de muitos meses e muitas conversas, percebi que o melhor caminho seria este.

Eu quero que pessoas comuns aprendam Django, portanto, este é o **livro eletrônico** que criei para essas pessoas comuns.

Pessoas comuns são pessoas que gostam de **passear com a família**, **ouvir música**, **conhecer gente legal** e **ganhar dinheiro**.

Tá certo que toda pessoa comum tem também lá suas esquisitices... então, o nosso ator principal aqui será o um alienígena.

Pronto, agora não temos mais problemas com a esquisitice!

Capítulo 1: Alatazan chega ao Planeta Terra

Há poucas semanas, Alatazan estava numa nave de transporte em massa vinda de **Katara**.

Katara (que significa **"nós resolvemos"** em seu idioma mais popular) é um planeta não muito distante, que gira em torno de um sol *gelado e azul claro**. Lá existe a tradição de enviar seus jovens para outros planetas, com o fim de passar um tempo ali aprendendo seus costumes e dando um tempo para os pais que querem ter uma folga das manias desses jovens.

Alatazan escolheu o Planeta Terra depois de consultar no Google Sky e achar bonitinha a forma rechonchuda, molhada e quentinha da Terra. E assim, veio parar aqui através de uma coisa que parece um prato cheio de lâmpadas coloridas. Sim, eles conseguem acesso à nossa internet, e você saberá em breve como fazem isso.

Algumas semanas depois de chegar à Terra, Alatazan percebeu que o nosso pequeno planeta estava em polvorosa com aquilo que chamávamos de **"Engenharia de Software"**. Ele ainda não compreendeu como haveria uma engenharia para uma coisa mole e sem forma, mas Alatazan é otimista, sempre. E sabe que vai resolver esse enigma.

Depois de resolver a questão de sua reluzente careca com uma peruca bacana (apesar do amarelo ser um tom de amarelo incomum para cabelos), Alatazan também procurou resolver uma questão menos importante mas muito chata: seus documentos.

Com documentos que o fizeram se tornar menos esquisito aos olhos dos terráqueos, Alatazan conseguiu uma vaga em um curso da tal "Engenharia de Software". Logo percebeu que não havia engenheiro de software algum, na verdade os profissionais da engenharia de software eram chamados de muitos nomes, como "analistas", "arquitetos", "projetistas", "desenvolvedores", "codificadores", "homologadores", "enroladores", enfim, eram muitos títulos, não muito claros, e Alatazan ainda não entendia bem para quê serviam tantos nomes para se realizar atividades tão parecidas.

Na saída da primeira aula, Alatazan estava um tanto atordoado com as confusas definições que ouvira. Ainda não entendia como Arquitetura e Engenharia serviam para definir quase as mesmas coisas de um assunto, e também não entendia pra quê serviam todos aqueles desenhos, que precisam ser refeitos todos os dias e repetidos em códigos e outras formas de repetições... Foi quando ele notou que uma garota e um rapaz eram os únicos que desciam a rampa, que pareciam se divertir.

Alatazan pensou: "puxa, do quê esses dois esquisitos estão rindo depois de toda essa tortura? Deve haver algo que eu preciso entender..."

Alatazan se achava no direito de achar os terráqueos esquisitos. Eram todos diferentes dele, e ele tinha um alto padrão de beleza para Katara: um belo topete (sim, isso também será explicado), olhos de desenho japonês, e dois belos dedos em cada pé!

E foi assim que Alatazan conheceu **Cartola** e **Nena**, seus parceiros nessa aventura.

Capítulo 2: O que é Django? Como é isso?

- Tzzzzt!

A campainha da casa de **Cartola** tocou, era Alatazan do lado de fora. Ele deu uma olhadela pela veneziana e correu para abrir a porta.

- Vem, estamos fazendo um projeto lá dentro, usando uma coisa nova, você vai gostar disso.

Para Alatazan todas as tecnologias terráqueas eram novas, mas a alegria de Cartola já indicava que ela parecia ser um pouco diferente das apresentadas no curso.

Correram pra dentro e **Nena** estava sentada ao seu laptop com alguns arquivos abertos e olhando o resultado de sua "arte" no navegador.

- Veja, o Cartola descobriu esse novo jeito de fazer sites. É uma tecnologia chamada Django... dá pra fazer muitas coisas diferentes sem muito esforço. O maior esforço é esquecer um pouco do que sabemos hoje pra entender o novo paradigma.

Mas enfim, o que é o Django?

Django é um framework, construído usando a linguagem Python.

Framework é uma **caixa de ferramentas**. Mas não é apenas isso, o Django é uma caixa de ferramentas **com um manual dentro**, e as ferramentas **combinam muito bem umas com as outras**. Você usa as ferramentas que quiser, e se você quiser substituir uma delas por outra, isso **é perfeitamente possível**.

Já o **Python** é uma linguagem de programação, trata-se de uma tecnologia que lê o código que você escreveu numa sintaxe conhecida e dá vida àquilo.

Mas o Python também não é só isso. Ele funciona em praticamente qualquer computador, seja com o Windows, Linux ou Mac, é fácil de aprender e de entender. Sua forma de tratar a programação é uma forma que **faz sentido**. Os caras que criaram o Python não tinham a menor vontade de complicar as coisas.

Além do Django e do Python, você vai precisar de um banco de dados.

O Banco de Dados é onde os dados são guardados. Quando você está em uma rodoviária ou aeroporto, existe o guarda-volumes, um local onde você paga um valor pra guardar suas coisas por um certo tempo. Quando precisa guardar dinheiro, você vai ao banco. O Banco de Dados é o lugar certo para guardar **dados**, e ele é independente do Django.

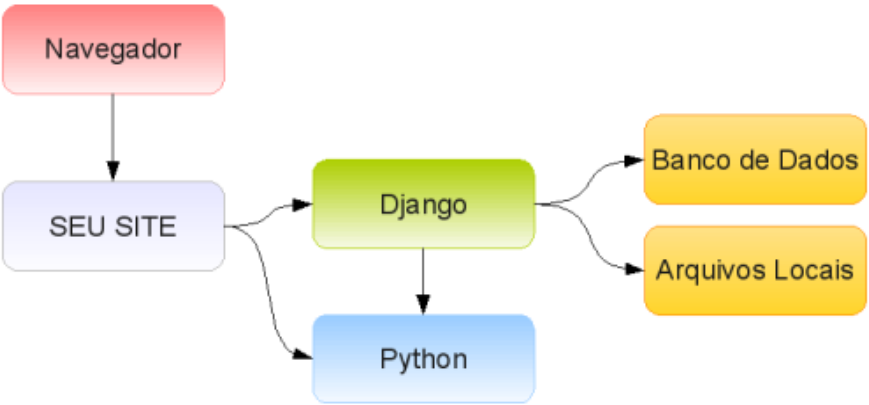
Os bancos de dados que o Django conhece são o **MySQL**, **PostgreSQL**, **SQLite**, **Oracle** e Microsoft **SQL Server**. Mas nós vamos trabalhar por um bom tempo só com o **SQLite**, que é o mais fácil deles.

E para ver o seu site ou programa funcionando, você vai precisar de um **Navegador**, que é o programa que você usa para acessar sites da web. Mozilla **Firefox**, Microsoft **Internet Explorer** e Apple **Safari** são alguns deles. Nós vamos usar o Firefox por ser o único destes que funciona tanto no Windows, quanto no Linux e no Mac. Mas não se preocupe com isso, o navegador é o que menos importa quando se trata de Django.

E como ele funciona?

Então resumindo a história, o Django funciona assim:

1. você tem um navegador, o Firefox por exemplo;
2. você entra no navegador e digita o endereço do seu site;
3. o site é feito em Django, usando a linguagem Python;
4. através do Django, seu site acessa dados do Banco de Dados e em arquivos locais e retorna para seu navegador uma bela página com funcionalidades em geral;
5. você olha sorridente para o navegador e vê o resultado final.



Alatazan agora sabe do que o Django se trata, e já se sentiu mais em casa. Em

Katara, as coisas costumam ser mais simples do que têm sido na Terra. Isso porque o povo de lá já passou por isso o suficiente pra descobrir que simplificar as coisas sempre ajuda no resultado final. O que eles não sabem é que simplificar evita cabelos brancos, já que eles não possuem cabelos.

No próximo capítulo, vamos baixar e instalar o Django, o Python e o que mais for necessário, e dar os primeiros passos para criar o primeiro projeto: um blog, que Alatazan vai usar para se comunicar com sua família.

Capítulo 3: Baixando e Instalando o Django

Quando Alatazan chegou à Terra, sua pequena nave foi enviada da nave maior, e ao chegar, chocou-se ao chão, meio desajeitada, e pareceu acertar alguém que passeava com seu cachorrinho.

Alatazan saiu de dentro um pouco desajeitado e uma velhinha lhe ofereceu um suco.

Ele adorou aquilo, e agora, na casa de Cartola, ele tomava um suco de tangerina pra refrescar um pouco antes de retomar à descoberta do Django.

Do nada, ele soltou essa:

- Quem inventou o suco?
- Sei lá, alguém que estava sentindo calor, há muito tempo atrás...
- Quem foi eu não sei, mas ainda bem que ele contou a receita pra alguém, senão teria ido dessa pra melhor com sua ideia guardada e deixando um monte de gente fadada ao cafezinho.
- A fruta é de preço baixo, fácil de comprar, a receita é livre para qualquer um, é só ter um pouquinho de tempo, água, liquidificador, açúcar, essas coisas... bom demais! Vou levar isso pra Katara e vou ficar rico fazendo sucos...

E depois dessas divagações, eles voltaram ao computador.

- Como é que se consegue o Django? - Alatazan já foi logo indagando ao Cartola...

Como conseguir o Django

O Django é **software livre**. O Python também é **software livre**.

Software livre é todo software que sua a receita é livre, pública, acessível para qualquer um. É como se você for a um restaurante e quiser ver como é a cozinha

pra ter certeza de que não estão lhe oferecendo gato por frango. É como se você quiser um dia fazer diferente e misturar suco de limão com abacate... parece estranho, mas você pode fazer isso se quiser (e é estranho que algumas pessoas realmente gostem).

Os caras que fizeram o Django acreditam que todo software (ou pelo menos a maioria deles) deveria ser livre. Eles ganham dinheiro usando software livre, mas não impedem que outras pessoas também o usem e ganhem dinheiro da mesma forma, ou até mais.

Mas se o software que você faz será livre ou não, isso também é um direito seu.

Então ficou fácil de resolver. Você pode baixar o Django e não pagar nada por isso. Usá-lo e não pagar nada por isso. E pode vender o seu trabalho com ele, e não repassar nada a eles por isso. Só é preciso que escreva em algum lugar que seu software foi escrito usando Django.

O mesmo vale para o Python e para as bibliotecas adicionais que o Django utiliza.

Baixando e instalando

Há várias formas de se instalar o Django. Algumas muito fáceis, outras muito educativas. Nós vamos seguir pelo caminho que deve ser o mais difícil deles. Isso é porque é importante passar por essas etapas para abrir sua mente ao universo Python e Django, mas caso queira o caminho mais prático, vá direto ao final deste capítulo e veja como fazer.

No Windows

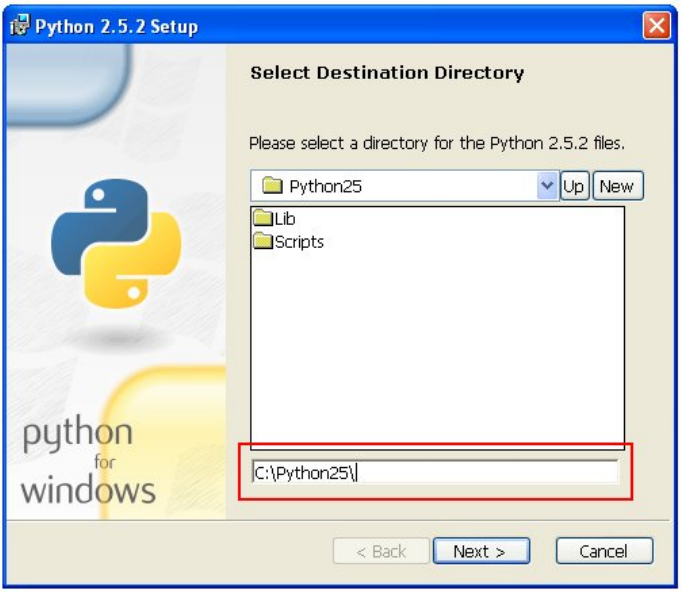
Bom, então vamos começar pelo **Python**.

Se você usa Windows, faça o download do instalador do Python, acessando a seguinte URL no seu navegador:

```
http://www.python.org/download/releases/2.5.2/
```

Após baixar o arquivo, clique duas vezes sobre ele e clique no botão **"Next"**.

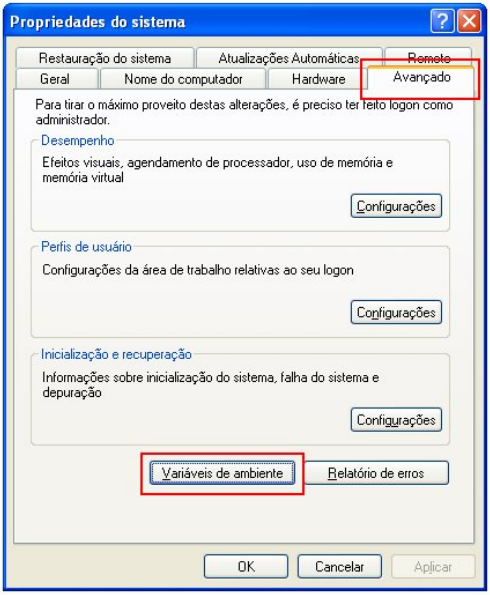
Na próxima página, a imagem abaixo será exibida, guarde o caminho indicado no campo (no caso da imagem abaixo, é **"C:\Python25\"**, pois vamos usá-lo logo após a instalação do Python. Depois dela, siga clicando no botão **"Next"** até finalizar.



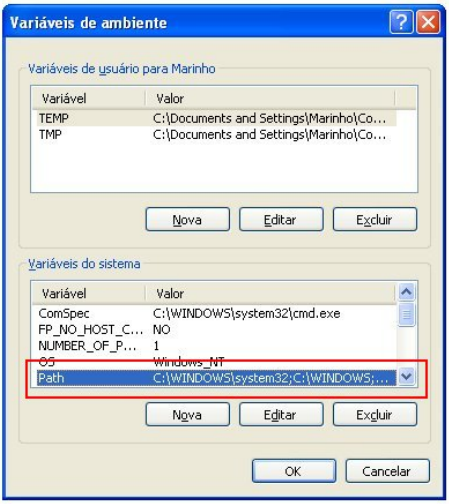
Por fim, é importante fazer isso: o Windows não consegue encontrar sozinho o Python, portanto, é preciso dizer a ele onde o Python está. Isso se faz adicionando o caminho de instalação do Python à variável de ambiente **PATH**, do Windows.

Pressione as teclas **Windows + Pause** (a tecla Windows é a tecla do logotipo do Windows).

Na janela aberta, clique na aba **"Avançado"** e depois disso, no botão *Variáveis de ambiente*:



Na caixa de seleção **"Variáveis do Sistema"** clique duas vezes sobre o item **"Path"**.



E ao final do campo **"Valor da Variável"**, adicione um ponto-e-vírgula e o caminho que você memorizou da instalação do Python, como está destacado abaixo.

Pronto! O Python está pronto para uso. Agora vamos ao **Django**!

Na página a seguir você encontrará alguns meios para instalar o Django.

```
http://www.djangoproject.com/download/
```

Vamos instalar o **tarball** da versão 1.0. Tarball é um arquivo compactado que termina com **".tar.gz"** e precisa de um programa de descompressão para ser aberto.

Para o **Windows**, o **7-Zip** é um software (também livre) que faz isso pra você. Faça o download da seguinte página e instale em sua máquina:

```
http://www.7-zip.org/
```

Voltando ao Django, localize **"Django-1.0.tar.gz"** na página de Download do Django citada acima. Faça o download do arquivo e descompacte onde bem quiser.

Dentro da pasta criada - que provavelmente vai se chamar **"Django-1.0"**) você vai criar um novo arquivo, chamado **"instalar.bat"**, edite usando o **Bloco de Notas** e escreva dentro o seguinte código:

```
python setup.py install
```

```
pause
```

Feche, salve, e clique duas vezes para executar.

Feito isso, o Django estará instalado. Agora vamos seguir para o último passo: **instalar a biblioteca do banco de dados SQLite**.

O SQLite é um banco de dados simples, sem muitos recursos. É ideal para o ambiente de criação e desenvolvimento. Não dá pra fazer muitas coisas com ele, mas você dificilmente vai precisar de fazer muitas coisas enquanto cria seu site. Para o Django trabalhar em conjunto com o SQLite, é preciso apenas instalar uma biblioteca, chamada **pySQLite**, e nada mais.

Então, vá até a página seguinte e faça o download do **tarball** (código-fonte, com extensão .tar.gz).

```
http://oss.itsystementwicklung.de/trac/pysqlite/#Downloads
```

Se a instalação do pySQLite apresentar **erros de compilação** no Windows, tente instalar com um dos instaladores executáveis disponíveis na página de download acima. Isso às vezes ocorre por consequência de algum tipo de incompatibilidade entre compiladores.

Após o download, faça o mesmo que fez com o Django:

1. descompacte usando o 7-zip
2. crie o arquivo **instalar.bat** com aquele mesmo código dentro
3. execute
4. pronto.

Pois bem, agora você tem um ambiente de desenvolvimento instalado para começar a usar.

No Linux, Mac e outros sistemas

Se você usa **Linux** ou **Mac**, os passos não serão diferentes dos seguidos para o Windows.

No site do Python são disponíveis os arquivos para download e instalação para esses sistemas. No caso do Django e pySQLite o processo é basicamente o mesmo, com as devidas diferenças comuns entre os sistemas operacionais.

Normalmente o Linux já vem com o Python e o pySQLite instalados.

Existe um caminho mais fácil?

Sim, existe um caminho bem mais fácil. Como já foi dito lá em cima, o processo acima é mais difícil, mas, o melhor para o seu aprendizado.

Se quiser instalar de forma mais fácil, use o **DjangoStack**. Veja neste site como fazê-lo:

> `http://marinhobrandao.com/blog/p/djangostack-facil-para-o-iniciante-aprender-django/`

Alatazan respirou aliviado. Apesar da sopa de letrinhas, a coisa não parecia tão complicada.

- então tudo se resume a instalar 3 coisas: o Python, o Django e o pySQLite.
- que são feitos de forma que os princípios do software livre sejam preservados: usando outros softwares livres - completou Nena
- e então, o que vamos fazer amanhã?
- vamos criar o seu primeiro projeto. Você já tem alguma idéia do que quer fazer?
- sim, quero dar um jeito de mandar mensagens para a minha família, mostrar como tem sido a minha vida por aqui, o que tenho descoberto, algumas fotos dos tênis da Rua 9...
- então o que você quer, é um **blog**. Vamos fazer isso então!

Capítulo 4: Criando um Blog maneiro

Era uma manhã fria, o vento de outono balançava as folhas das poucas árvores na rua onde Alatazan morava, e alguém deixou ali um envelope engraçado, **triangular**. Para Alatazan, o envelope não era engraçado, era **exatamente igual** aos envelopes mais usados em Katara.

Ele abriu o envelope e tirou o objeto prateado reluzente, com cheirinho de novo. Uma das pontas do envelope estava ligeiramente amassada, mas não tinha problema, pois o papel eletrônico era chamado de papel exatamente por ser **flexível**.

Na imagem animada do papel, sua mãe esbravejava por sua falta de consideração. Desde sua chegada, Alatazan não havia enviado sequer um "alô", mas mal sabia sua mãe que por aqui não havia uma agência do sistema interestelar de correios.

Mas Katara tem um jeito de acessar a nossa Internet, e ele teve então a ideia de criar um blog.

Um blog é o jeito mais simples de escrever sobre sua vida. Toda vez que dá vontade de escrever, o sujeito vai lá, **cria um título e escreve** o que quiser, e a página mostra sempre os últimos itens que ele escreveu.

Porquê um blog?

Um blog é fácil e simples de se fazer. E também simples de criar novidades.

Hoje vamos deixar o **Blog** funcionando, a seguir vamos esclarecer umas coisas, melhorar outras, colocar os **RSS**, depois vamos aos **Comentários**, e assim por diante. Cada pequeno passo de uma vez, e bem resolvido.

Mas antes de tudo, precisamos resolver uma questão ainda em aberto...

Escolhendo um editor

O Django não se prende a um editor específico. Existem centenas deles, alguns com mais recursos, outros mais simples. É como modelos de tênis, cada um usa o

seu, e há aqueles que preferem sapatos, sandálias, ou andar descalços mesmo. Há ainda aqueles não preferem, mas ainda assim o fazem.

O editor que você escolhe hoje, pode ser descartado amanhã, e o próximo também. Não se trata de um casamento, mas sim de uma escolha do momento, até que você encontra aquele que se encaixa com o seu perfil.

Então vamos começar pelo **Bloco de Notas**, e depois vamos conhecer outros.

Ok, agora vamos criar o projeto

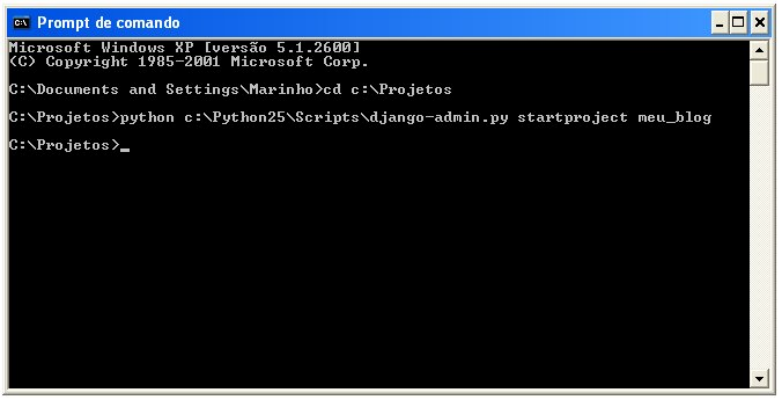
Antes de mais nada, vamos criar uma pasta para colocar nossos projetos: **c:\Projetos** (no Linux ou Mac: **/Projetos**).

E agora abra o **Prompt de comando dos MS-DOS** (no Linux ou Mac é o **Console** ou **Terminal**) e localize a pasta criada com:

```
cd c:\Projetos
```

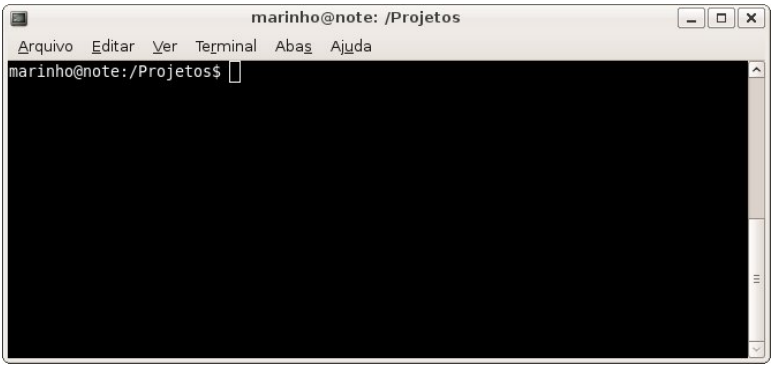
Depois disso, digite o comando que cria projetos:

```
python c:\Python25\Scripts\django-admin.py startproject meu_blog
```

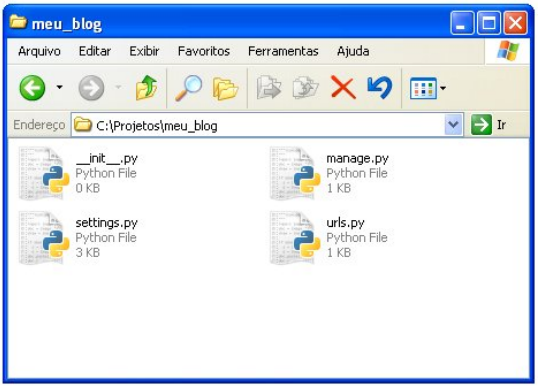


No Linux ou Mac é um pouquinho diferente:

```
$ cd /Projetos
$ django-admin.py startproject meu_blog
```



Pronto, agora feche essa janela e vamos voltar à pasta do projeto, em **c:\Projetos\meu_blog**



Agora, vamos ver o site funcionando (mesmo que sem nada - ou com quase nada - dentro). Na pasta do projeto, crie um arquivo chamado **"executar.bat"** e o edite com o **Bloco de Notas**. Dentro dele, escreva o seguinte código:

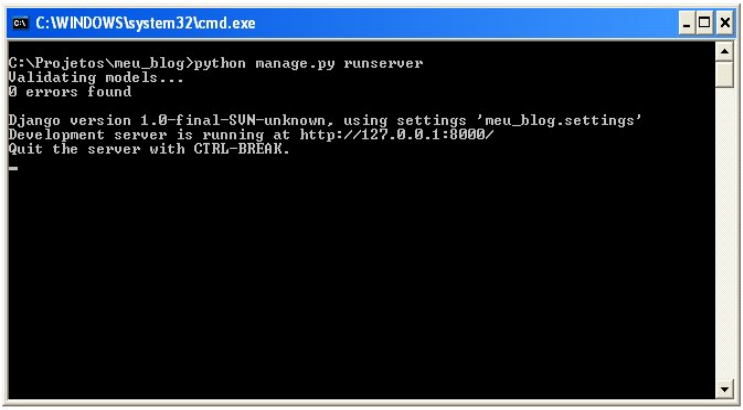
```
python manage.py runserver
pause
```

Nota: no **Linux** e no **MacOSX** o equivalente aos arquivos **.BAT** são os arquivos **.sh**. A diferença é que eles não suportam o comando **"pause"** e devem iniciar com a seguinte linha:

```
#!/bin/bash
```

Você deve também definir a permissão de execução ao arquivo usando o comando **chmod +x** ou usando a janela de Propriedades do arquivo.

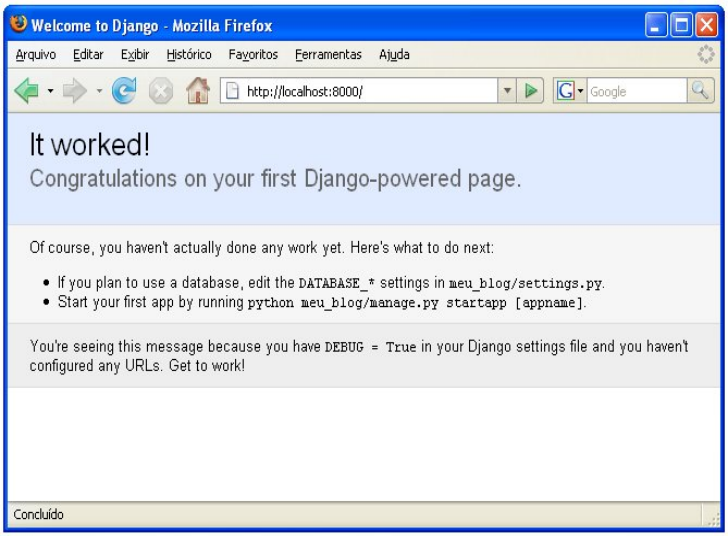
Salve o arquivo. Feche o arquivo. Execute o arquivo clicando duas vezes sobre ele.



Abra seu **navegador** - o **Mozilla Firefox**, por exemplo - e localize o endereço:

```
http://localhost:8000/
```

E é isso que você vê:



Mas isso ainda não é nada. Você ainda não possui nada em seu projeto. Ou pelo menos, nada que alguém possa ver.

O arquivo settings do projeto

Agora na pasta do projeto, edite o arquivo **settings.py** com o **Bloco de Notas** e faça as seguintes modificações:

1. A linha que começa com **DATABASE_ENGINE**, deve ficar com **DATABASE_ENGINE = 'sqlite3'**

2. A linha que começa com **DATABASE_NAME**, deve ficar com **DATABASE_NAME = 'meu_blog.db'**

3. E logo abaixo da linha que possui **'django.contrib.sites',** acrescente outra linha com **'django.contrib.admin',**

O arquivo de URLs do projeto

Salve o arquivo. Feche o arquivo. Agora edite o arquivo **urls.py**, também com o **Bloco de Notas**, fazendo as seguintes modificações:

1. Na linha que possui **# from django.contrib import admin**, remova o "# " do início - **não se esqueça de remover o espaço em branco**

2. Na linha que possui **# admin.autodiscover()**, remova o "# " do início

3. Na linha que possui **# (r'^admin/(.*)', admin.site.root),**, remova o "# " do início

Salve o arquivo. Feche o arquivo. Agora vamos gerar o banco de dados (isso mesmo!).

Geração do banco de dados

Na pasta do projeto (meu_blog), crie um arquivo chamado **gerar_banco_de_dados.bat** e escreva o seguinte código dentro:

```
python manage.py syncdb
pause
```

Salve o arquivo. Feche o arquivo. Execute o arquivo, clicando duas vezes sobre ele.

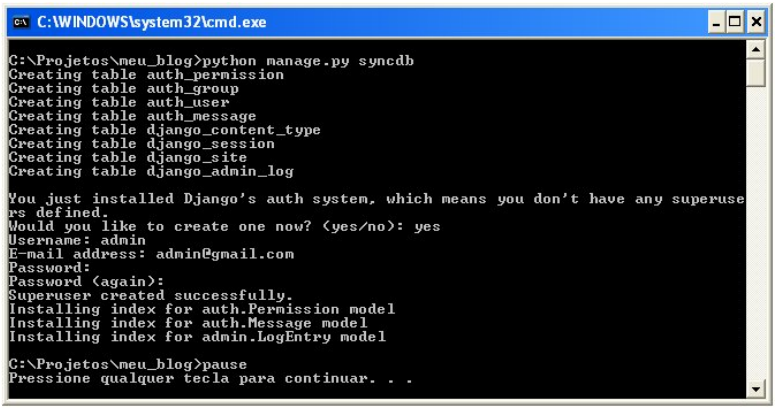
A geração do banco de dados cria as tabelas e outros elementos do banco de dados e já **cria também um usuário de administração do sistema**. Isso acontece

porque naquele arquivo **settings.py** havia uma linha indicando isso (calma, logo você vai saber qual).

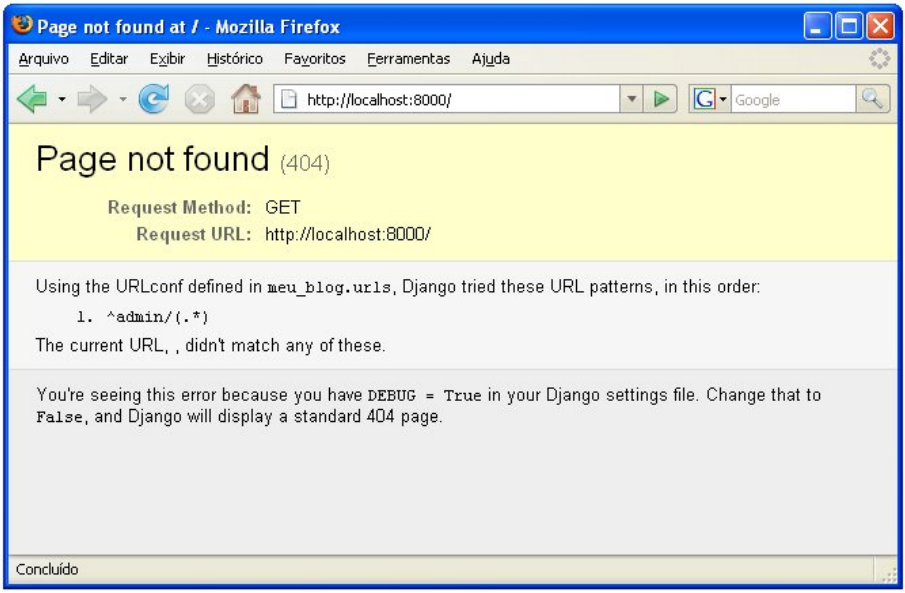
Sendo assim, ele pergunta pelo nome desse usuário (**username**), seu **e-mail** (se você digitar um e-mail inválido, ele não vai aceitar) e a **senha**.

Assim, para ficar mais fácil, informe os dados assim:

- Username: **admin**
- E-mail: **admin@gmail.com**
- Password: **1**
- Password (again): **1**



Ao exibir a frase **Pressione qualquer tecla para continuar. . .**, feche a janela volte ao navegador, **pressione F5** pra ver como ficou.



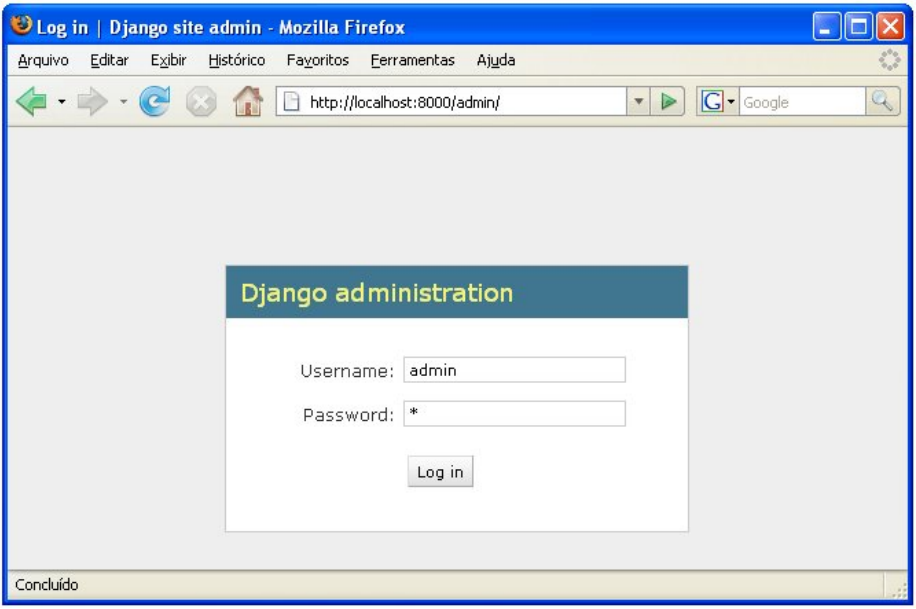
Isso aconteceu porque agora você possui alguma coisa em seu projeto. E quando você possui *"alguma coisa"*, qualquer outra coisa que você não possui é considerada como **Página não encontrada**. Então, que *coisa* é essa?

A interface de administração

Na barra de endereço do navegador, localize o endereço:

```
http://localhost:8000/admin/
```

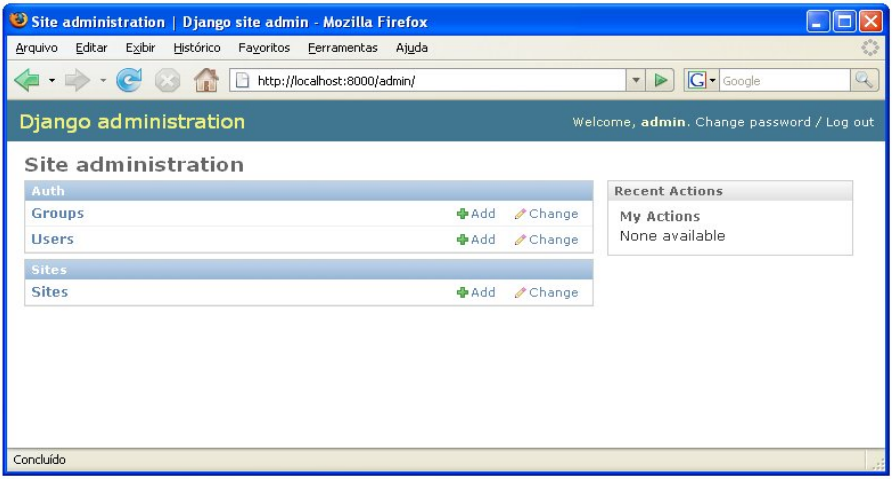
E a seguinte página será carregada, para autenticação do usuário.



Você vai informar o usuário e senha criados no momento da geração do banco de dados:

- Username: **admin**
- Password: **1**

E após clicar em **Log in**, a seguinte página será carregada:



Criando a primeira aplicação

Agora, vamos criar uma aplicação, chamada **"blog"**, ela vai ser responsável pelas funcionalidades do blog no site. Portanto, crie uma pasta chamada **"blog"** e dentro dela crie os arquivos abaixo:

- __init__.py
- models.py
- admin.py

Abra o arquivo **models.py** com o **Bloco de Notas** e escreva o seguinte código dentro:

```
from django.db import models


class Artigo(models.Model):
    titulo = models.CharField(max_length=100)
    conteudo = models.TextField()
    publicacao = models.DateTimeField()
```

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **admin.py** com o **Bloco de Notas** e escreva dentro:

```
from django.contrib import admin
from models import Artigo
admin.site.register(Artigo)
```

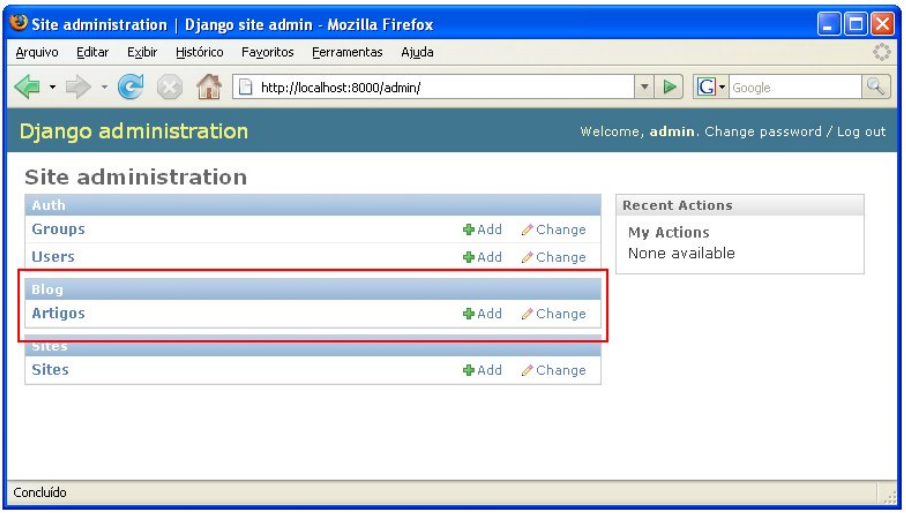
Salve o arquivo. Feche o arquivo.

Vamos voltar à pasta do projeto (**meu_blog**), e editar novamente o arquivo **settings.py**, fazendo o seguinte:

1. Abaixo da linha que possui **'django.contrib.admin',** acrescente outra linha com **'blog',**

Salve o arquivo. Feche o arquivo.

E agora volte ao navegador, pressionando a tecla **F5**:

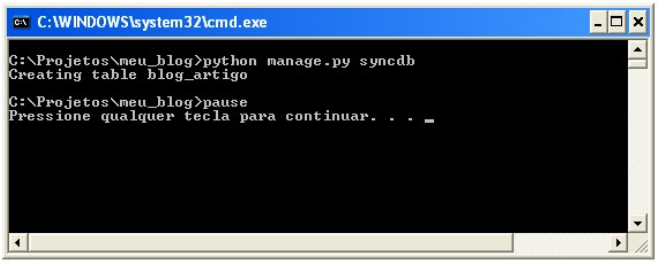


Opa! Agora vemos ali uma coisa nova: uma caixa da nova aplicação **"blog"**.

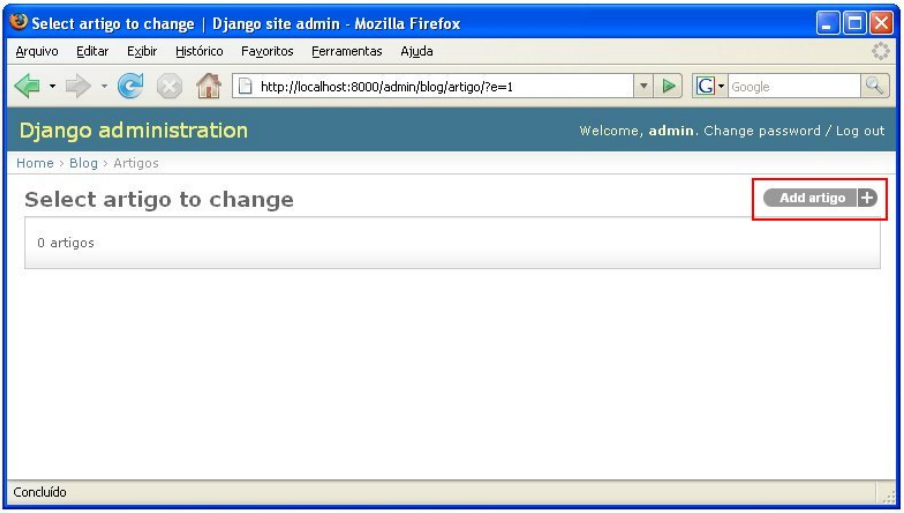
Mas como nem tudo são flores, ao clicar no link **Artigos**, será exibido um erro, de banco de dados.

Isso acontece porque você não gerou o banco novamente, depois de criar a nova aplicação. Então vamos fazer isso.

Clique duas vezes sobre o arquivo **gerar_banco_de_dados.bat** e veja o resultado a seguir:

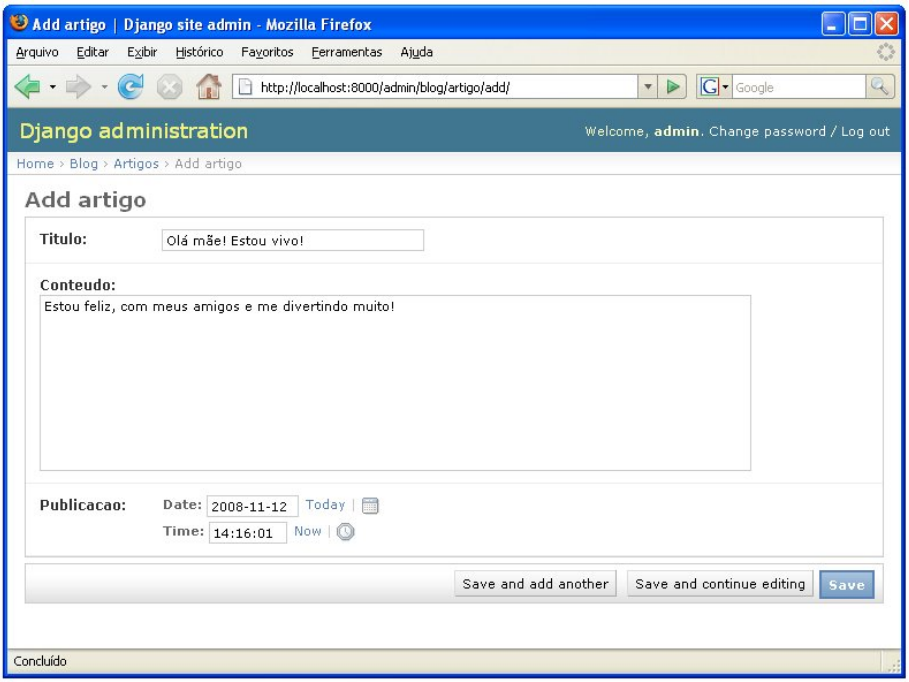


Agora, ao voltar ao navegador e atualizar a página com **F5**, a página será carregada da maneira desejada:



Vamos inserir o primeiro artigo?

Clique sobre o link **"Add artigo"** e preencha as informações, clicando por fim no botão **"Save"** para salvar.



Bom, já temos a interface de administração do blog funcionando, agora vamos ao mais interessante!

Saltando da administração para a apresentação

Bom, temos a interface de administração do blog funcionando, mas isso ainda não é suficiente, porque os visitantes do blog precisam ter uma visão **mais agradável** e limitada dessas informações, e no caso da mãe de Alatazan, multiplique isso por dois.

Pois então vamos agora fazer uso do poder das **generic views** e dos **templates** do Django para criar a página onde vamos apresentar os artigos que o Alatazan escrever.

Vamos então começar criando **URLs** para isso.

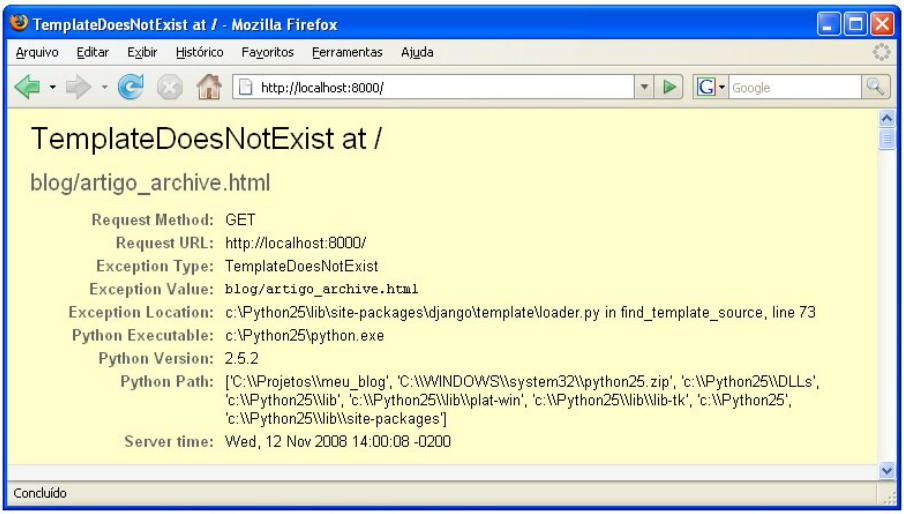
URLs são os endereços das páginas do site. Um exemplo é a **http://localhost:8000/admin/**, apontada para a interface de administração.

Ao tentar carregar a URL **http://localhost:8000/** em seu navegador, será exibida aquela página amarela com **Página não encontrada**, se lembra? Pois então agora vamos criar uma página para responder quando aquela URL for chamada.

Na pasta do projeto (**meu_blog**), abra o arquivo **urls.py** para editar, e modifique para ficar da seguinte maneira:

```
from django.conf.urls.defaults import *


# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()


from blog.models import Artigo


urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
      {'queryset': Artigo.objects.all(),
       'date_field': 'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
)
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, e pressione **F5** para atualizar a página, e o resultado será como este abaixo:



Isso acontece porque ainda falta um **arquivo em HTML** para exibir as informações do **Blog**, que chamamos de **Template**.

Vamos criá-lo?

Ok, abra a pasta **blog**, contida na pasta da projeto (onde já estão os arquivos **__init__.py**, **models.py** e **admin.py**), crie uma pasta chamada **"templates"** e dentro dela crie outra pasta, chamada **"blog"**. Por fim, dentro da nova pasta criada, crie novo arquivo chamado **"artigo_archive.html"** e escreva o seguinte código dentro:

```
<html>
<body>


<h1>Meu blog</h1>


{% for artigo in latest %}
<h2>{{ artigo.titulo }}</h2>


{{ artigo.conteudo }}
{% endfor %}
```

```
</body>
</html>
```

Este é um HTML simples, que percorre a lista de **artigos** do **blog** e exibe seu **Título** e **Conteúdo**.

Atenção: neste momento, é importante fechar a janela do **MS-DOS** (**Console** ou **Terminal**, no Linux ou Mac) onde o Django está rodando, e executar novamente o arquivo **executar.bat**. Isto se faz necessário, pois é a primeira vez que adicionamos um template no site, o que obriga que todo o Django seja iniciado novamente do zero.

Veja como ficou:

Bom, então, é isso!

Finalizando por hoje...

Aos poucos Alatazan vai percebendo que a cada novo passo que ele dá, as explicações detalhadas de Cartola e, especialmente Nena, vão se tornando desnecessárias, pois ele vai formando o conhecimento necessário para fazer aquelas tarefas mais comuns.

No entanto, é muito claro pra ele que há muitas coisas não explicadas (ou pouco explicadas) ali. Foi aí que ele se virou para Nena e disse:

- Err... fascinante, mas, eu ainda estou sem entender a maior parte das coisas... como por exemplo aquele...

- Calma meu amigo, deixa de ser fominha, uma coisa de cada vez - interferiu o

sorridente Cartola, batendo em seu ombro e deixando cair um dos fones do ouvido.

• Vamos respirar um pouco, amanhã vamos explicar o porquê de cada coisa, e você vai compreender cada um desses arquivos criados, editados, modificados, executados, enfim. Mas isso só depois de um bom suco! - Nena se empolgou com a vontade de Alatazan, que se empolgou com a empolgação de Nena. Cartola já era empolgado por natureza.

Capítulo 5: Entendendo como o Django trabalha

Quando ainda era criança, Alatazan andava pela rua alegremente quando zuniu uma bola de metal cheia de braços perto de sua cabeça.

Na verdade não eram muitos braços, eram apenas dois mesmo.

A bola de metal era uma espécie de robô de modelo antigo. Seus circuítos pareciam não funcionar muito bem, mas ele emanava um sorriso arregalado, num jeito espalhafatoso que Alatazan gostou.

Um homem gritou:

- Vai **Ballzer**! Vai **Ballzer**!

O robô seguiu seu caminho, voando a menos de 2 metros de altura em direção à praça.

Na praça havia uma mesa comprida. Antes da mesa estava um **menino vestido de despachante**, segurando um carimbo, e sentados ao longo da mesa haviam robôs de modelos diferentes, cada um com alguns apetrechos. Um deles tinha objetos de barbearia, outro tinha colas, outro tinha tintas com pincéis, e assim por diante.

Ballzer entregou um cubo de borracha ao menino, ele gritou algo como "seção 42!" e passou o cubo ao primeiro robô da mesa, que fez alguma coisa no cubo e o passou ao próximo robô, que o cheirou e não fez nada, depois passou adiante.

Do outro lado da mesa havia uma mulher alta, magra e de pele rosada que recebeu o cubo do último robô, cheirou, abriu, tirou um papel de dentro e leu, depois guardou, pegou **uma coisa** em sua bolsa e devolveu ao robô, que refez o ritual às avessas, passando a coisa para o próximo robô e assim até chegar ao **Ballzer**, que voltou em velocidade para entregar a coisa a um senhor do outro lado da rua, próximo do muro.

Alatazan ouviu o senhor dizer:

- Muito bem **Ballzer**, muito bem.

Isso se repetiu por mais vezes. Às vezes o senhor sorria, às vezes não. Pegava outro cubo, entregava ao Ballzer e assim se seguia...

Eles estavam jogando **Bhallz**, um complexo jogo muito praticado nas escolas para adolescentes.

Voltando ao Django...

Na web, as coisas são parecidas.

Imagine que o senhor do outro lado da rua é o **seu usuário**, usando o navegador.

O menino é o **URL dispatcher**.

Os robôs à mesa são **middlewares**.

A moça rosada, ao final da mesa, é uma **view**.

O cubo é uma **requisição** (ou **HttpRequest**).

A coisa que a moça devolveu aos robôs é uma **resposta** (ou **HttpResponse**).

E por fim, o maluco e imprevisível **Ballzer**, é a própria **Internet**.

Ou seja, cada qual à sua vez, fazendo seu papel na linha de produção, para **receber um pedido do usuário e retornar uma resposta equivalente**, que na maioria das vezes, é uma página bonita e colorida.

Entendendo a requisição

Na requisição (ou **HttpRequest**) há diversas informações. Uma das informações é a URL, que é composta pelo protocolo, o domínio, a porta (às vezes), o caminho e os parâmetros (às vezes também).

Exemplo:

```
http://localhost:8000/admin/?nome=Mychell&idade=15
```

- Protocolo: **http**
- Domínio: **localhost**
- Porta: **8000**
- Caminho: **/admin/**
- Parâmetros: **nome = Mychell** e **idade = 15**

Quando a porta não é citada, então você deve entender que se trata da **porta 80**,

a porta padrão do protocolo **HTTP**.

Outras informações que se encontram na requisição são aquelas sobre o navegador e computador do usuário, como seu **IP**, **sistema operacional**, **idiomas suportados**, **cookies** e diversas outras coisas.

O Handler

A primeira coisa que acontece quando a requisição chega ao Django, está no handler. Essa parte **nunca é vista ou notada por você**, mas é o primeiro passo antes da requisição chegar aos middlewares, e também é o último passo antes da resposta voltar do Django para o usuário final.

Middlewares

Middlewares são pequenos trechos de código que analisam **a requisição na entrada e a resposta na saída**. A requisição é analisada por eles.

Você pode determinar quais middlewares estarão na fila à mesa para analisar a requisição e pode até criar os seus próprios middlewares.

Um dos middlewares faz com que toda a **segurança e autenticação de usuários** seja feita. Outro adiciona a função de **sessão**, uma forma de memorizar o que o usuário está fazendo para atendê-lo melhor. Há ainda outro middleware, que memoriza as respostas no **Cache**, pois se caso a próxima requisição tenha o mesmo **caminho** e os mesmos **parâmetros**, ele retorna a resposta memorizada, evitando o trabalho de ser processada novamente pela **view**.

Sendo assim, após passar por todos esses middlewares, a requisição passa a ter outras informações, como **qual usuário está autenticado**, **sessão atual** e outras coisas.

O URL Dispather

Depois de passar pelos middlewares, a requisição é analisada pelo URL Dispatcher, um importante componente do Django que verifica o **endereço** - especialmente a parte do **caminho** - e verifica o arquivo **urls.py** do projeto para apontar qual **view** será chamada para dar a resposta.

Ah, a view

A view é uma função escrita em Python, e na maioria das vezes, escrita por você.

Ela faz isso: recebe uma requisição (**HttpRequest**) e retorna uma resposta

(**HttpResponse**).

É aqui que entra seu trabalho diário, que é analisar uma requisição, fazer algo no **banco de dados** e retornar uma página.

O Django possui algumas *views* prontas para coisas que sempre funcionam do mesmo jeito. Elas são chamadas **Generic Views** e estão espalhadas por toda parte. Há *generic views* úteis para blogs, para cadastros em geral, para lembrar senha, autenticar, sair do sistema, redirecionar, enfim, há muitas delas.

Você não é obrigado a usar generic views. Elas estão lá para você usá-las se assim o desejar.

As views escritas por você devem ser escritas em um arquivo chamado **views.py**.

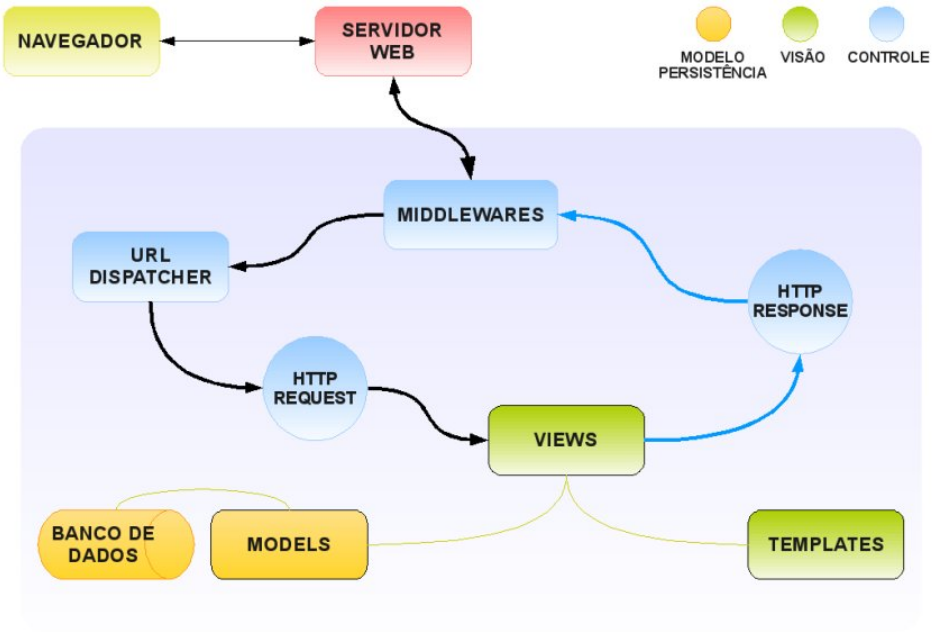
Os arquivos de templates

As páginas no Django são em geral guardadas em **Templates**, que são arquivos HTML (ou outro formato) que possuem sua própria lógica em interpretar àquilo que lhes é passado e no final renderizar um HTML, que é passado como resposta ao usuário. No entanto, esta não é uma regra, pois muitas vezes não será retornado um HTML como resposta. Mas isso será muito mais esclarecido nos próximos capítulos.

A camada de Modelo

Para armazenar e resgatar informações do **banco de dados**, não é necessário ir até ele e conhecer a linguagem dele (o bom, velho, e cada vez mais sumido **SQL**). Você usa uma ferramenta do Django chamada **ORM**, que interpreta o seu código, leva aquilo até o banco de dados, e depois devolve as informações desejadas.

A parte do código onde você configura quais são seus modelos de dados e que tipo de informações eles devem armazenar, é um arquivo chamado **models.py**.



Então, tudo se resume em MVC

MVC é a sigla que resume tudo isso:

- Modelo (Model)
- Visão (View)
- Controle (Controller)

Modelo é onde estão as definições dos dados, como eles devem ser armazenados e tratados. É lá que você diz quais campos uma tabela deve ter, seus tipos e valores padrão e se eles são obrigatórios ou não. Dentre outras coisas.

Visão são as funções que **recebem requisições e retornam respostas**, ao usuário, a outro computador, a uma impressora ou qualquer outra coisa externa. Ou seja, as views.

E **Controle** são todas as coisas que ficam no meio do caminho, como o **handler**, os **middlewares** e o **URL dispatcher**. A maior parte dessas coisas é feita pelo próprio Django, e você deve se preocupar pouco ou nada com isso.

E onde entram o **banco de dados** e os **templates**?

O banco de dados é a camada de **persistência**, e não faz parte do MVC, pois não faz parte do Django em si.

Os templates são arquivos utilizados pelas views, são apenas auxiliares, como uma bolsa utilitária ao lado, que você usa se quiser ou se precisar.

Passando adiante...

- Fascinante tudo isso, não? Meu pai conta que nas fábricas de software mais antigas trabalhavam com outro conceito, chamado **Cliente/Servidor**, mas isso nunca caiu bem na Web, era muito lento...

Os olhos de Nena até brilhavam. Ela adorava essas coisas de conceitos e arquitetura.

- Bom, não é muito diferente de jogar **Bhallz**. Cada pedacinho tem um papel simples a cumprir. Complicado é pegar a coisa toda de uma vez, mas se você jogar numa posição de cada vez, vai ver que é fácil de jogar... - respondeu um Alatazan mais aliviado.

Cartola estava um pouco disperso, entretido com seu *gadget*. Ele o havia configurado para receber os podcasts do **This Week In Django** sempre que houvesse novidades, e havia acabado de chegar um episódio novo.

No próximo capítulo, vamos colocar o **RSS** para funcionar no Blog, e entender como é fácil levar suas novidades às pessoas sem que elas precisem vir até você.

Capítulo 6: O RSS é o entregador fiel

Cartola é obcecado por informação: lê notícias de política, esportes e economia, sites com artigos sobre robótica, receitas de comida senegalesa e blogs de nerds de renome.

Nena não deixa por menos, mas ela gostava mesmo é de entretenimento. Acessa blogs de celebridades, ao mesmo tempo que curte cada novidade sobre metodologias ágeis.

Alatazan percebeu que seria complicado concorrer o precioso tempo de seus leitores com essa gente toda, então logo ele notou que precisa mudar seu blog para publicar seus textos em formato **RSS**.

RSS não se trata da sociedade conservadora mais risonha de Katara. RSS é somente um formato que segue os padrões do XML, mas que foi criado para carregar informações como **Título**, **Data de Publicação** e **Conteúdo** (dentre outras) de artigos. Qualquer site que lance esse tipo de informação, pode publicar seu conteúdo também em forma de RSS e assim seus leitores usam programas como o **Google Reader** para serem avisados sempre que houver novidades.

RSS também é muito útil para **Podcasts** e ainda para **exportar e importar** esse mesmo tipo de informação.

E claro, o Django já vem com isso pronto, sem dor e com bons recursos!

Então vamos lá?

Para trabalhar com RSS no Django, vamos antes ajustar algumas coisas que serão necessárias daqui em diante. E o melhor momento para fazer esses ajustes, é agora.

Ajustando settings fundamentais do projeto

No Django, um **site** é fruto de um **projeto**, que é composto de algumas (ou

muitas) **aplicações**.

Uma aplicação é um conjunto - de preferência não extenso - de funcionalidades com foco em resolver uma questão específica. Um **Blog** por exemplo, é uma aplicação.

Todo projeto possui um arquivo chamado **settings.py**, onde são feitas configurações que determinam a sua essência.

Uma coisa importante para ajustar nas settings do projeto, é a **TIME_ZONE**. É nesta setting que determinamos em que fuso horário o projeto deve estar adequado. Há uma lista desses fusos horários no seguinte site:

```
http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
```

Na pasta do projeto (**meu_blog**), abra o arquivo **settings.py** no editor e localize a seguinte linha:

```
TIME_ZONE = 'America/Chicago'
```

Agora modifique para ficar assim:

```
TIME_ZONE = 'America/Sao_Paulo'
```

Este é apenas um exemplo. Caso não esteja na zona de fuso horário da cidade de São Paulo, vá até o site citado acima, encontre o seu fuso horário, e ajuste a **TIME_ZONE** para código respectivo.

Salve o arquivo. Feche o arquivo.

Determinando a ordenação dos artigos do Blog

Agora, vamos à pasta da aplicação **blog**. Nesta pasta, abra o arquivo **models.py** com o editor e localize a seguinte linha:

```
class Artigo(models.Model):
```

Modifique esse trecho do código de forma que fique assim:

```
class Artigo(models.Model):

    class Meta:

        ordering = ('-publicacao',)
```

Essa modificação faz com que a listagem dos artigos seja sempre feita pelo campo **"publicacao"**. Ali há um detalhe importante: o **sinal de menos** (-) à esquerda de 'publicacao' indica que a **ordem deve ser decrescente**.

Localize esta outra linha:

```
publicacao = models.DateTimeField()
```

E modifique para ficar assim:

```
publicacao = models.DateTimeField(
       default=datetime.now,
       blank=True
       )
```

Isso faz com que não seja mais necessário informar o valor para o campo de publicação, pois ele assume a **data e hora atuais** automaticamente. O argumento **blank** indica que o campo **pode ser deixado em branco**, já que ele vai assumir a data e hora atuais nesse caso.

Porém, isso não basta.

O elemento **datetime** não está embutido automaticamente no seu arquivo, e você deve trazê-lo de onde ele está, para que o Python o use da forma adequada. Do contrário, o Python irá levantar um erro, indicando que o objeto **datetime** não existe.

Portanto, acrescente a linha a seguir no início do arquivo, na primeira linha:

```
from datetime import datetime
```

Dessa forma, o arquivo **models.py** fica assim:

```
from datetime import datetime
from django.db import models


class Artigo(models.Model):
    class Meta:
         ordering = ('-publicacao',)


    titulo = models.CharField(max_length=100)
    conteudo = models.TextField()
    publicacao = models.DateTimeField(
      default=datetime.now,
      blank=True
      )
```

Algumas coisas sobre Python serão esclarecidas daqui a alguns capítulos. Mas por hora, é importante ressaltar que você deve **sempre respeitar a edentação**. O padrão mais indicado no Python é que seus blocos sejam sempre edentados a cada **4 espaços**, portanto, muito cuidado com a tecla de tabulação. Não é a mesma coisa.

Muito bem. Feitos esses ajustes, vamos ao que realmente interessa.

Aplicando RSS dos artigos do blog

A primeira coisa a compreender aqui, é que o Django é feito por pequenas partes. E a maior parte dessas pequenas partes não são ativadas quando você cria um projeto.

Esse é um cuidado essencial para que seu projeto fique **o mais leve e compacto possível**, carregando para a memória e processando somente aquilo que é necessário. E claro, quem sabe o que é necessário, **é você**.

As aplicações especiais que já vêm com o Django, são chamadas de **contribs**, e aquela que faz o trabalho pesado do RSS é chamada de **syndication**.

Vamos então editar novamente o arquivo **settings.py** da pasta do projeto, para adicionar a aplicação **syndication** às aplicações do projeto.

Localize a linha abaixo:

```
INSTALLED_APPS = (
```

No Python, toda expressão que é divida entre sinais de vírgula (,) e contido entre parênteses, é chamada de **Tupla**. Trata-se de uma lista que não será modificada enquanto o projeto estiver em execução.

INSTALLED_APPS é uma tupla, onde indicamos todas as aplicações do projeto. Todos os itens contidos ali terão um tratamento especial no seu projeto, indicando **classes de modelo de dados** (tabelas para o banco de dados), **views** e outras informações contidas em seus arquivos **models.py**, **views.py**, **admin.py**, dentre outros.

Vamos portanto, adicionar ao final da tupla a aplicação **'django.contrib.syndication'**, a nossa estrela principal aqui. Portanto, agora a tupla vai ficar assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',


    'blog',
)
```

Não é necessário que a ordem seja exatamente esta, mas é recomendável que se separe as aplicações **contribs** das demais aplicações, sempre que possível.

Salve o arquivo. Feche o arquivo. Agora vamos adicionar uma **URL** para este nosso **RSS**.

Localize o arquivo **urls.py** da pasta do projeto e abra-o para edição.

Neste arquivo, cada **tupla** informada dentro da função **patterns()** indica um padrão de URL. Veja como está abaixo:

```
urlpatterns = patterns('',
   (r'^$', 'django.views.generic.date_based.archive_index',
 {'queryset': Artigo.objects.all(),
  'date_field': 'publicacao'}),
   (r'^admin/(.*)', admin.site.root),
)
```

Portanto, vamos adicionar a nossa nova URL após a última tupla, para ficar assim:

```
urlpatterns = patterns('',
   (r'^$', 'django.views.generic.date_based.archive_index',
 {'queryset': Artigo.objects.all(),
  'date_field': 'publicacao'}),
   (r'^admin/(.*)', admin.site.root),


 (r'^rss/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
    {'feed_dict': {'ultimos': UltimosArtigos}}),
)
```

A tupla adicionada indica que a nova URL vai exibir os artigos em formato RSS na seguinte endereço:

```
http://localhost:8000/rss/ultimos/
```

Não se esqueça da importância dessa palavra **ultimos**, pois é ela que identifica o nosso RSS (sim, podemos ter diversos RSS diferentes em um mesmo site).

Mas ainda não é suficiente, pois o **UltimosArtigos** é um estranho ali. Se você rodar o sistema agora, vai notar uma mensagem de erro, pois esse elemento **UltimosArtigos** realmente não existe.

Pois então, na linha superior à iniciada com **urlpatterns**, vamos adicionar a seguinte linha:

```
from blog.feeds import UltimosArtigos
```

E agora todo o arquivo **urls.py**, depois das modificações, fica assim:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

from blog.models import Artigo
from blog.feeds import UltimosArtigos

urlpatterns = patterns('',
   (r'^$', 'django.views.generic.date_based.archive_index',
 {'queryset': Artigo.objects.all(), 'date_field': 'publicacao'}),
   (r'^admin/(.*)', admin.site.root),
 (r'^rss/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
    {'feed_dict': {'ultimos': UltimosArtigos}}),
)
```

Pois bem. Salve o arquivo. Feche o arquivo.

Vamos seguir para a derradeira tarefa antes de ver o nosso RSS funcionando: na pasta da aplicação **blogs**, crie um novo arquivo chamado **feeds.py**, e escreva o seguinte código dentro:

```
from django.contrib.syndication.feeds import Feed

from models import Artigo

class UltimosArtigos(Feed):
    title = 'Ultimos artigos do blog do Alatazan'
    link = '/'

    def items(self):
             return Artigo.objects.all()

    def item_link(self, artigo):
             return '/artigo/%d/'%artigo.id
```

Pronto! Antes de entender em detalhes o que fizemos no código acima, vamos

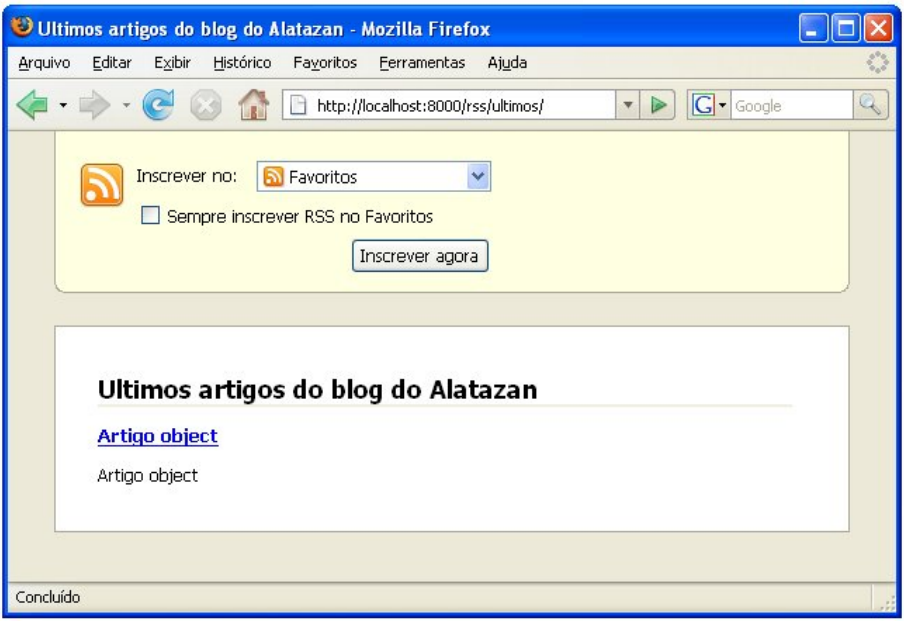
ver o resultado disso!

Primeiro, execute o seu projeto, clicando duas vezes sobre o arquivo **executar.bat** da pasta do projeto.

Agora localize o seguinte endereço em seu navegador:

```
http://localhost:8000/rss/ultimos/
```

E o resultado será este:



Gostou do que viu? Não... eu ainda não. Veja que o **título** e a **descrição** do artigo estão como **"Artigo object"**. Não está como nós queremos. Vamos lá resolver isso?

Na pasta da aplicação **blog**, há uma outra pasta, chamada **templates**. Dentro dela, você vai criar mais uma pasta, chamada **feeds**.

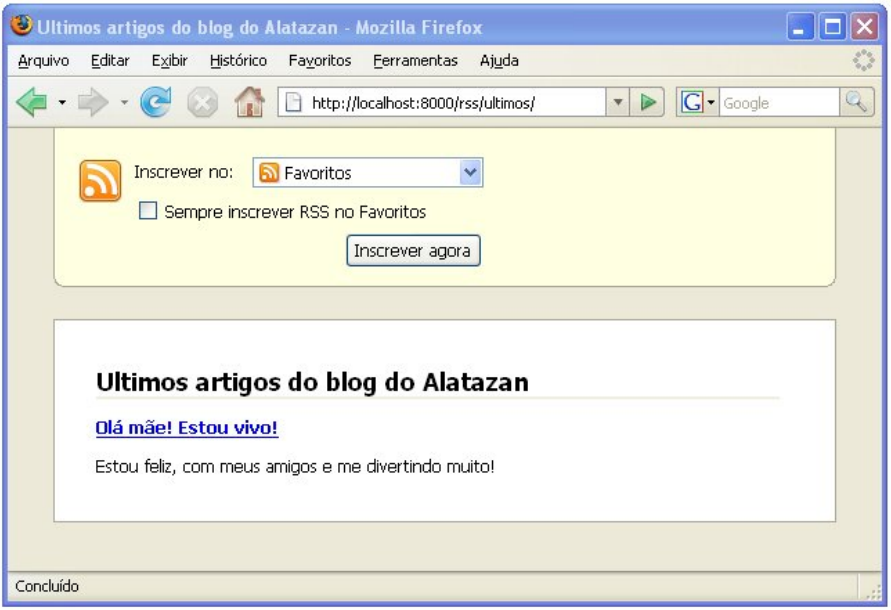
Dentro da nova pasta, vamos criar um template que será usado para determinar o **título** de cada artigo. Ele será chamado **ultimos_title.html**. Esse nome se deve à soma do nome desse RSS (**ultimos**) com **_title.html**, o nome padrão para isso que queremos fazer. Escreva o seguinte código dentro:

```
{{ obj.titulo }}
```

Salve o arquivo. Feche o arquivo. Crie um novo arquivo, chamado **ultimos_description.html**, e escreva o seguinte código dentro:

```
{{ obj.conteudo }}
```

Salve o arquivo. Feche o arquivo. Vá ao navegador e pressione a tecla **F5** para ver como ficou.



Que tal agora? Está melhor?

Ok, vamos conhecer um pouco mais do que fizemos.

O atributo **title** recebeu o valor **'Ultimos artigos do blog do Alatazan'**, que é título deste RSS.

O atributo **link** recebeu o valor **'/'**, indicando que o endereço na web equivalente a este RSS é o do **caminho** '/', que somado ao protocolo ('http'), domínio ('localhost') e porta ('8000'), fica assim:

```
http://localhost:8000/
```

O método **def items(self)** carrega uma **lista de artigos**. A linha **return Artigo.objects.all()** indica que este método vai retornar no RSS **todos** os objetos da classe **Artigo** (em ordem decrescente pelo campo **publicacao**, lembra-se?).

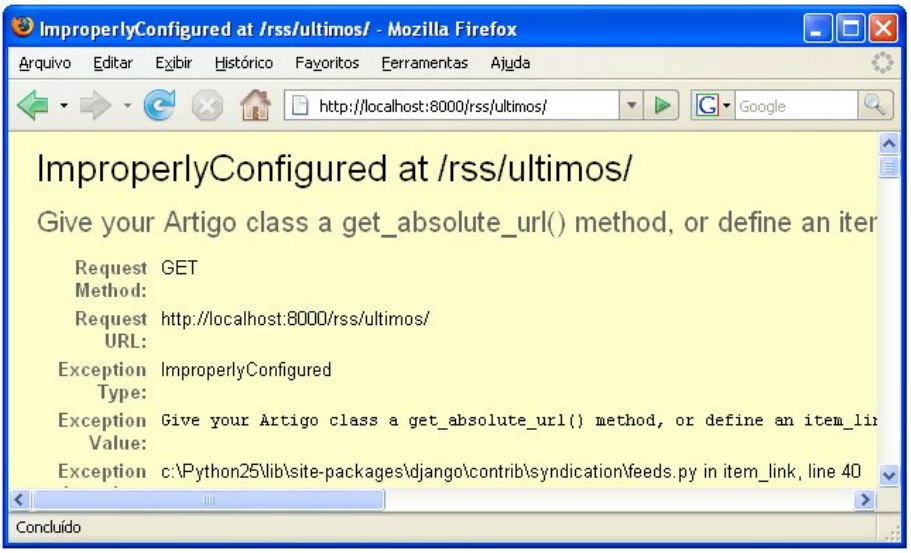
E o método **def item_link(self, artigo)** retorna, para cada **artigo**, seu endereço na web, que definimos para ser o caminho '/artigo/' somado ao **id** do artigo, mais uma barra (/) ao final, que no fim da história vai ficar assim, por exemplo, para um artigo de id **57**:

```
http://localhost:8000/artigo/57/
```

No entanto, nós podemos fazer isso de uma forma diferente. Faça assim: remova as seguintes linhas do arquivo **feeds.py**:

```
def item_link(self, artigo):
    return '/artigo/%d/'%artigo.id
```

Salve o arquivo. Feche o arquivo. Após um **F5** no navegador, na URL do RSS, veja o resultado:



A mensagem de erro completa é esta:

Give your Artigo class a get_absolute_url() method, or define an item_link() method in your Feed class.

Veja que ele sugere a criação do método **item_link** (que acabamos de remover) ou do método **get_absolute_url()** na classe **Artigo**. Vamos fazer isso?

Na pasta da aplicação **blog**, abra o arquivo **models.py** para edição e ao final da classe **Artigo** (o final do arquivo) adicione as seguintes linhas de código:

```
def get_absolute_url(self):
    return '/artigo/%d/'%self.id
```

Ou seja, agora o arquivo **models.py** todo vai ficar da seguinte forma:

```
from datetime import datetime
from django.db import models
```

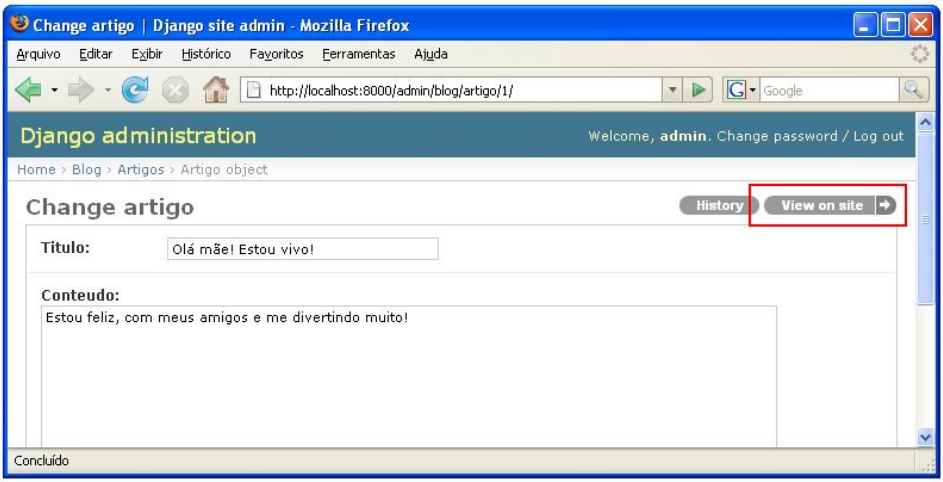
```
class Artigo(models.Model):
    class Meta:
             ordering = ('-publicacao',)

    titulo = models.CharField(max_length=100)
    conteudo = models.TextField()
    publicacao = models.DateTimeField(
      default=datetime.now,
      blank=True
      )

    def get_absolute_url(self):
         return '/artigo/%d/'%self.id
```

Salve o arquivo. Feche o arquivo. De volta ao navegador, atualize a página com **F5** e veja que voltamos ao normal!

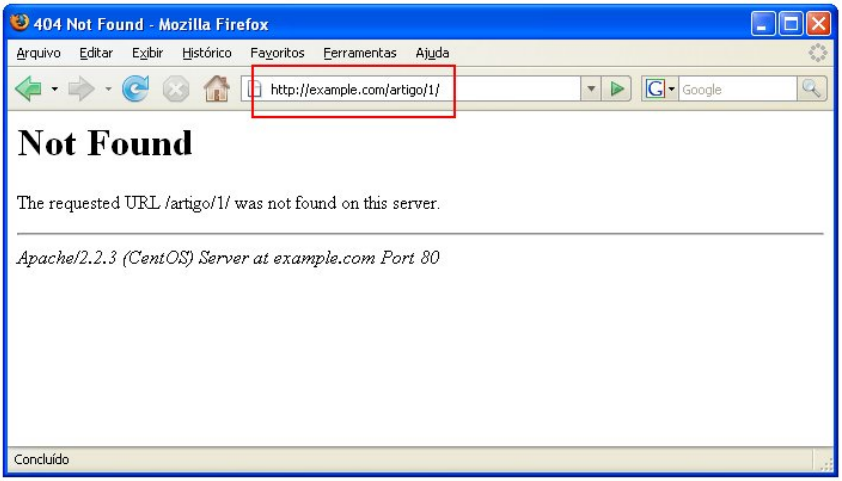
O que fizemos agora foi definir um **endereço absoluto de URL** para **artigo**, ou seja, cada artigo tem o seu endereço, e esse endereço é útil não apenas para o RSS, mas também para outros fins, Veja por exemplo o seguinte endereço:

```
http://localhost:8000/admin/blog/artigo/1/
```

O resultado é:



Notou o link **"View on site"** em destaque no canto superior direito? Pois bem, clique sobre ele, e o resultado será:



Hummm... não saiu como queríamos... Mas isso aconteceu pelo seguinte motivo:

O Django possui suporte a **múltiplos sites em um único projeto**. Isso quer dizer que se você tem um projeto de blog, você pode ter os blogs do Cartola, do Alatazan e da Nena, **todos eles usando o mesmo projeto**, ainda que trabalhem com

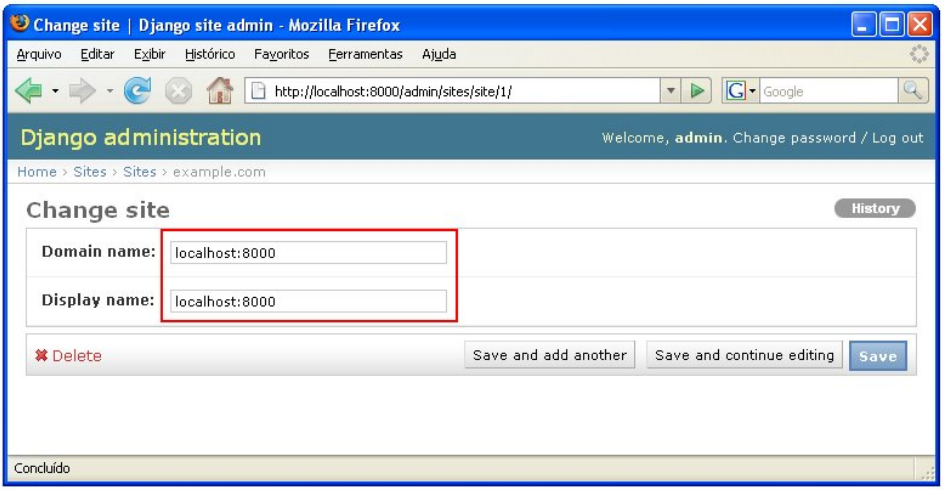
informações distintas. Isso às vezes é muito útil.

Quando um projeto é criado, ele recebe um **"site"** padrão, que tem o endereço **http://example.com**, justamente para você modificar e deixar como quiser. Então vamos mudar isso para **http://localhost:8000/**, certo?

Carregue o endereço do **admin** no navegador:

```
http://localhost:8000/admin/
```

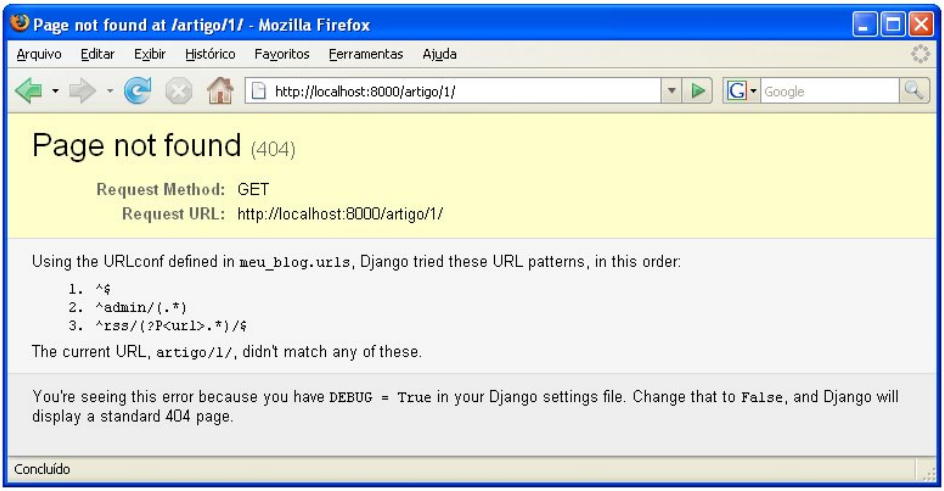
Na caixa **'Sites'**, clique no item **'Sites'**. Na página carregada, clique sobre **'example.com'**, e modifique, substituindo **'example.com'** para **'localhost:8000'**, assim:



Clique no botão **Save**.

Feche a janela de execução do Django e execute novamente o arquivo **executar.bat** da **pasta do projeto**.

Volte à URL anterior (http://localhost:8000/admin/blog/artigo/1/), clique em **"View on site"** e o resultado será este:



Bom, já conhecemos esta página, certo? Isso quer dizer que essa página não existe.

Esta página será a **página de um artigo específico**. Ou seja, Alatazan tem um blog, e na página principal do blog são exibidos todos os artigos, porém, cada artigo possui sua própria URL - sua própria página. Então vamos criá-la?

Na pasta do projeto, abra o arquivo **urls.py** para edição e adicione a seguinte URL:

```
(r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
```

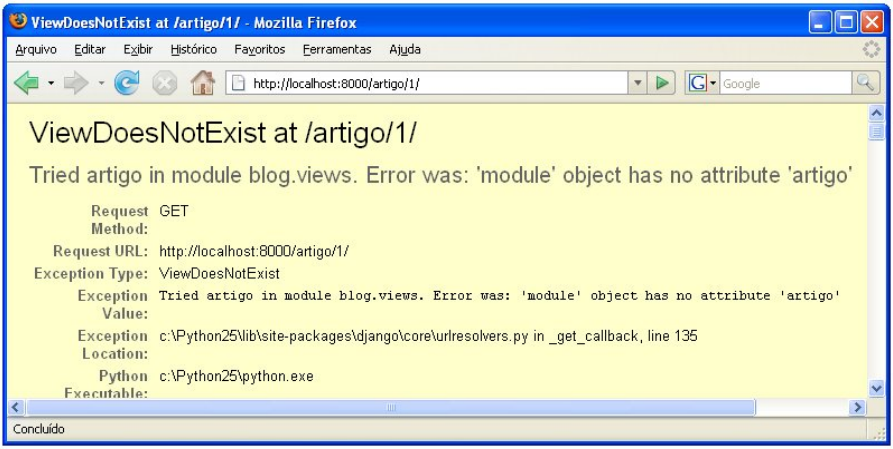
Ou seja, o arquivo **urls.py** vai ficar assim:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

from blog.models import Artigo
from blog.feeds import UltimosArtigos

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
```

```
        'date_field': 'publicacao'}
     ),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
)
```

Salve o arquivo. Feche o arquivo.

Volte ao navegador, pressione a tecla **F5** e observe que a mensagem mudou:

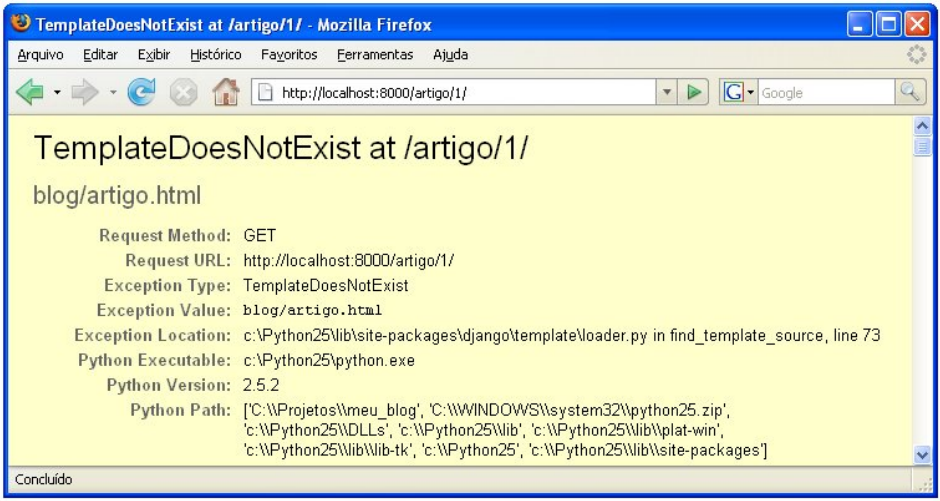


Essa mensagem indica que a URL informada (http://localhost:8000/artigo/1/) realmente existe, mas que não foi encontrada a view **'blog.views.artigo'**. Isso acontece porque a view realmente não existe. Ainda!

Agora, na pasta da aplicação **blog**, crie um novo arquivo, chamado **views.py** e escreva o seguinte código dentro:

```
from django.shortcuts import render_to_response


def artigo(request, artigo_id):
    return render_to_response('blog/artigo.html')
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, pressione **F5** e o resultado já mudou:

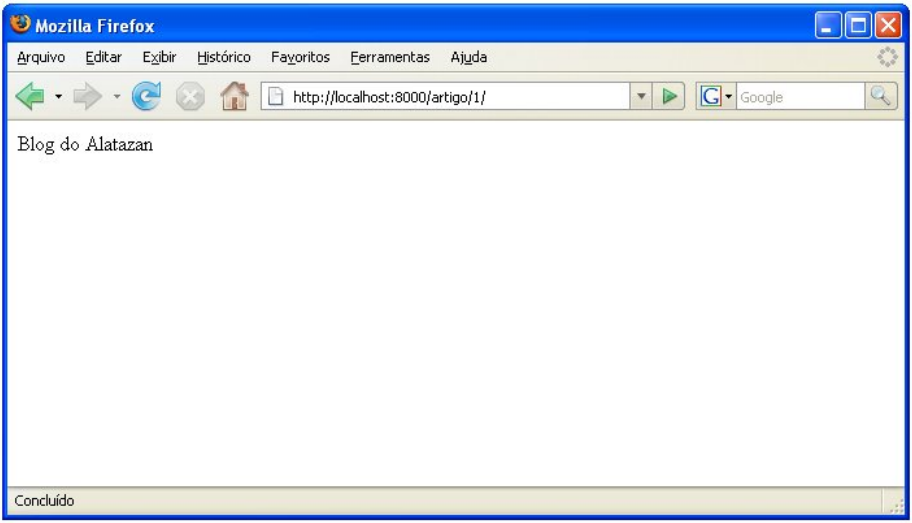


A mensagem diz: **"O template 'blog/artigo.html' não existe"**. Sim, não existe, mas vamos criá-lo!

Na pasta da aplicação **blog**, abra a pasta **templates**, e dentro dela, a pasta **blog**. Crie um novo arquivo chamado **'artigo.html'** e escreva o seguinte código dentro:

```
<html>
<body>


Blog do Alatazan


<h1>{{ artigo.titulo }}</h1>


{{ artigo.conteudo }}


</body>
</html>
```

Agora volte ao navegador, atualize a página com um **F5** veja o que tem:



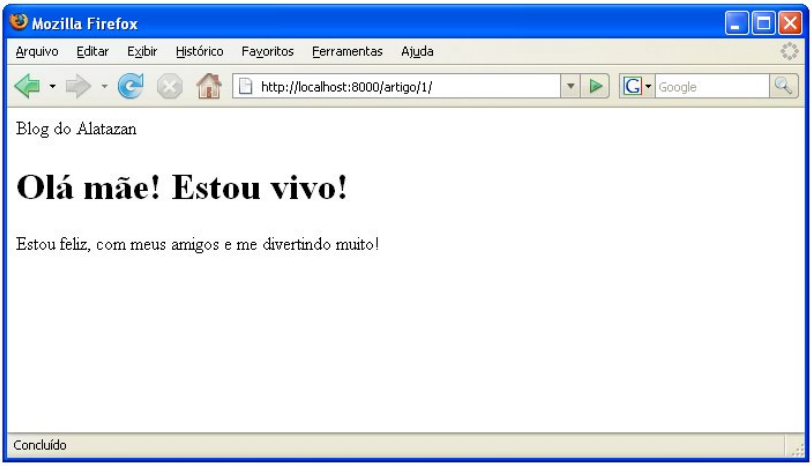
Bom, aconteceu algo errado aqui, não? Veja que no arquivo do template, nós indicamos as variáveis **{{ artigo.titulo }}** e **{{ artigo.conteudo }}** para exibirem respectivamente o **título** e o **conteúdo** do artigo. Mas tudo o que apareceu foi 'Blog do Alatazan'. Porquê?

Porque para as variáveis informadas serem válidas, é preciso que alguém indique que elas existem, e de onde elas vêm.

As variáveis válidas para os templates são indicadas a uma parte "oculta" do processo, chamada **Contexto**. Para indicar a variável **artigo** para o contexto deste template, vamos modificar a **view**. Portanto, na pasta da aplicação **blog**, abra o arquivo **views.py** para edição e o modifique, para ficar da seguinte forma:

```
from django.shortcuts import render_to_response


from models import Artigo


def artigo(request, artigo_id):
    artigo = Artigo.objects.get(id=artigo_id)
    return render_to_response('blog/artigo.html', locals())
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, e... **shazan**



Pronto. Temos uma página para o artigo.

Mas antes de terminarmos o capítulo, falta fazer uma coisa importante: **deixar o navegador saber que o nosso blog fornece RSS**.

Para fazer isso, na pasta da aplicação **blog**, carregue a pasta **templates** e dali a pasta **blog**. Abra o arquivo **artigo_archive.html** para edição, e modifique, para ficar assim:

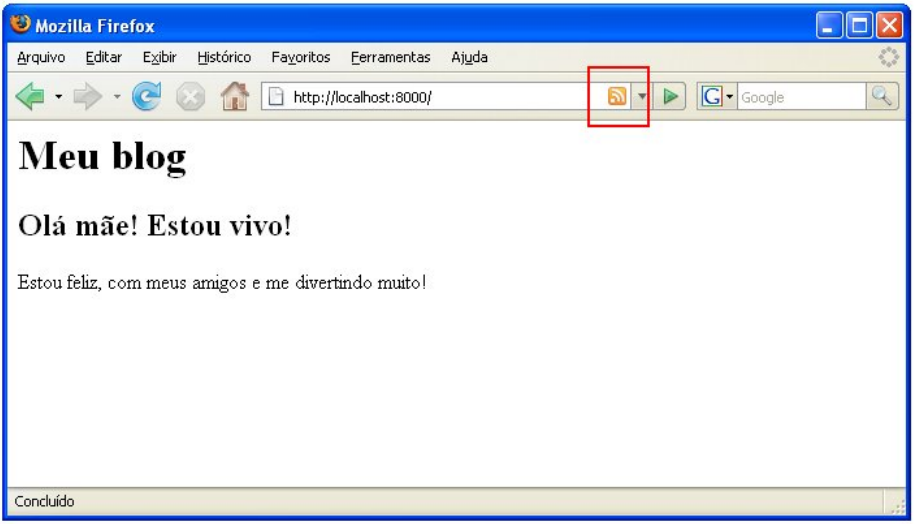
```
<html>

<head>

<link rel="alternate" type="application/rss+xml" title="Ultimos
artigos do Alatazan" href="/rss/ultimos/" />

</head>


<body>


<h1>Meu blog</h1>


{% for artigo in latest %}

<h2>{{ artigo.titulo }}</h2>


{{ artigo.conteudo }}

{% endfor %}
```

```
</body>
</html>
```

Salve o arquivo. Feche o arquivo. Vá ao navegador e carregue a página principal de nosso site:

```
http://localhost:8000/
```

O resultado será este:



Observe o **ícone do RSS** na barra de endereço.

Fechando o dia com sabor de dever cumprido

- Cara, dá pra ver sua empolgação... pegou tudo? - disse Cartola também agitado.
- É fascinante! Com essa aulinha de hoje, olha só o que eu aprendi:
  - Mudar o fuso horário do projeto
  - Mudar a ordenação dos artigos
  - Dar um valor automático para a data do artigo
  - Adicionar uma aplicação de contrib
  - Criar a URL do RSS
  - Criar a classe de Feed do RSS

- Criar as templates para o título e a descrição dos artigos no RSS
- Criar **a URL e a view do artigo** e sua template
- Mudar a URL padrão do site
- Informar variáveis ao contexto do template
- Fazer o ícone do RSS aparecer no navegador
- e...

• Calma! Você vai ter um ataque aqui hein... - interferiu **Nena** - só não vá se esquecer de que **no Python, edentação correta é obrigação** viu... não vá esquecer aquele espaço vazio à esquerda, senão *é pau na certa*. Trocar espaço por TAB também é fria...

Os dois corações de Alatazan pulsavam forte e os três caíram na gaitada, rindo da situação.

• Olha, vamos cada um pra sua casa que agora eu vou dar uma volta por aí. Esse blog está feínho que dói, amanhã vamos **dar um jeito no visual dele**.

Cartola disse isso e já foi ajeitando a roupa pra seguir com a sua vida.

Capítulo 7: Fazendo a apresentação do site com templates

Alatazan levantou cedinho, correu até a geladeira e buscou um doce de goiaba. Aquilo era muito bom e ele devorava cada pedacinho, como se fossem os deliciosos lagartos que vendem nas panificadoras de sua cidade natal.

Ele ligou a TV. Sempre achou muito engraçada aquela forma primitiva de entretenimento.

No programa educativo daquela manhã, estavam apresentando como funciona a linha de fabricação e montagem de automóveis.

Qualquer um sabe - isso, aqui na Terra, claro - que um automóvel é movido pela força de seu motor, que faz um (ou ambos) de seus eixos girar, e isso faz as rodas levarem o carro à frente. Há diversas outras partes fundamentais em um veículo, e elas são compostas uma por uma, até que no fim da linha de montagem, passam a ser feitos ajustes, pinturas e encaixes com um propósito também muito importante: **o conforto do motorista e o visual do carro**.

Não há nada de agradável em dirigir um automóvel com cadeiras de ferro ou observar um carro sem os devidos acabamentos.

A situação do blog de Alatazan é mais ou menos essa: **funciona mas está longe de ser uma beleza**.

Então vamos trabalhar nos templates!

Um dos elementos mais importantes do Django, é seu sistema de templates. Ele possui uma forma peculiar de trabalhar que facilita muito a criação da apresentação de suas páginas, mas os caras que criaram o Django fizeram questão que esse trabalho não se misture a outras coisas, ou seja, **não é interessante trazer para dentro do template aquilo que deveria ter sido feito no código da view**.

Mas chega de conversa e vamos ao que interessa!

Antes de mais nada, execute o seu projeto, clicando duas vezes no arquivo **executar.bat** da pasta do projeto (**meu_blog**).

No navegador, carregue a página inicial:

```
http://localhost:8000/
```

Observe por alguns segundos:

Você levaria a sério um site assim? Alatazan também não.

Então vamos criar um **CSS** para melhorar isso.

CSS significa **Cascading Style Sheet**.

Por alguma razão mórbida, Alatazan se lembrou da sociedade RSS de Katara, que se refere a "Rarams Sarabinis Sarados". O significado do nome dessa sociedade conservadora e risonha não era conhecido nem mesmo por seus seguidores, já que era conservada da mesma forma (e por sinal desorganizada) há tantos milênios que o significado se perdeu por lá.

Alatazan voltou sua atenção ao CSS e notou que isso não tinha nada a ver com aquilo. E prosseguiu.

Aproveitando o devaneio de Alatazan, a partir de agora, para facilitar a nossa comunicação, vamos usar o caminho de pastas de forma mais compacta, certo?

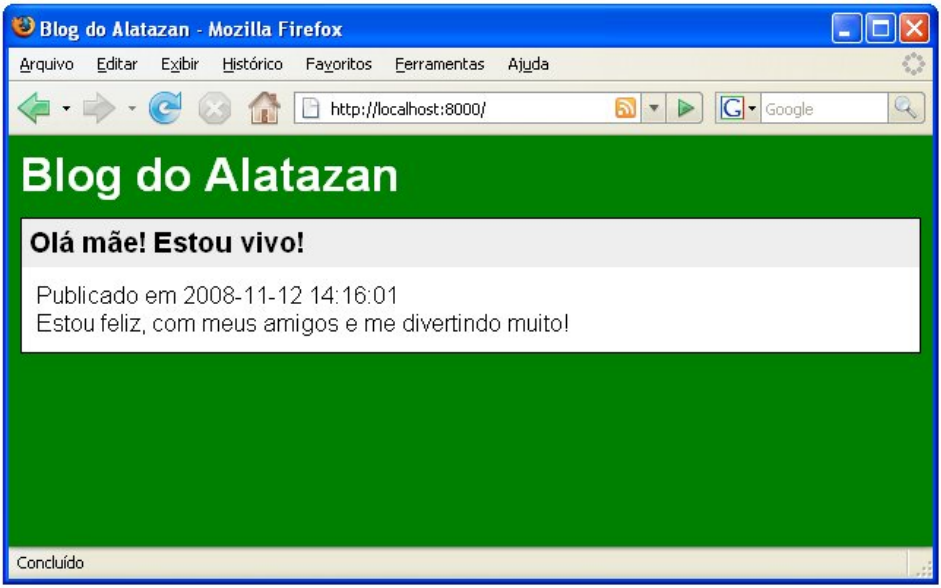
Partindo da pasta do projeto (**meu_blog**), em **blog/templates/blog** abra o arquivo **artigo_archive.html**, e o modifique de forma que fique assim:

```
<html>
```

```
<head>
<title>Blog do Alatazan</title>
<link rel="alternate" type="application/rss+xml" title="Ultimos
artigos do Alatazan" href="/rss/ultimos/" />
<style type="text/css">
body {
  font-family: arial;
  background-color: green;
  color: white;
}

h1 {
  margin-bottom: 10px;
}

h2 {
  margin: 0;
  background-color: #eee;
  padding: 5px;
  font-size: 1.2em;
}

.artigo {
  border: 1px solid black;
  background-color: white;
  color: black;
}

.conteudo {
  padding: 10px;
}
</style>
</head>
```

```
<body>

<h1>Blog do Alatazan</h1>

{% for artigo in latest %}
  <div class="artigo">
    <h2>{{ artigo.titulo }}</h2>

    <div class="conteudo">
      Publicado em {{ artigo.publicacao }}<br/>
      {{ artigo.conteudo }}
    </div>
  </div>
{% endfor %}

</body>
</html>
```

No navegador, atualize a página com **F5** e veja como ficou:

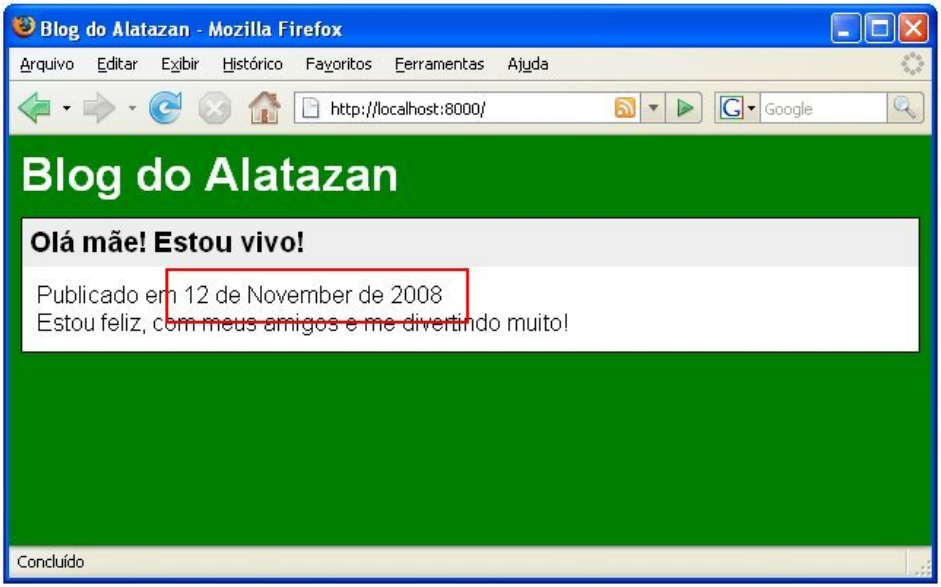


Opa, visual já melhorou! Mas aquela data de publicação... não ficou boa.

A maioria das modificações foram estéticas, usando **CSS e HTML**.

A exceção foi a data de publicação, que trata-se da referência **{{ artigo.publicacao }}**, que indica que naquele lugar do HTML, será exibido o conteúdo do atributo **publicacao** da variável **artigo**. Mas ele ficou definitivamente desagradável.

Há uma forma de melhorar isso, usando um **template filter**, chamado **"date"**. Troque o **{{ artigo.publicacao }}** por **{{ artigo.publicacao|date:"d \d\e F \d\e Y" }}** e veja como fica:



O trecho de código que você acrescentou (**|date:"d \d\e F \d\e Y"**), faz o seguinte: recebe uma data (ou data/hora, como é este caso) e faz o tratamento, exibindo no seguinte formato: **"DIA de MÊS de ANO"**, e retorna esse valor formato em seu lugar - sem modificar a variável original.

Acontece que o mês está em inglês. Então isso quer dizer que o todo o site está preparado para trabalhar na língua inglesa, o que não é o caso de Alatazan, que está vivendo no Brasil.

Mudando o idioma padrão do projeto

Então vamos agora editar o arquivo **settings.py** da pasta do projeto para mudar isso. Pois então, com o arquivo **settings.py** aberto em seu editor, localize o seguinte

trecho de código:

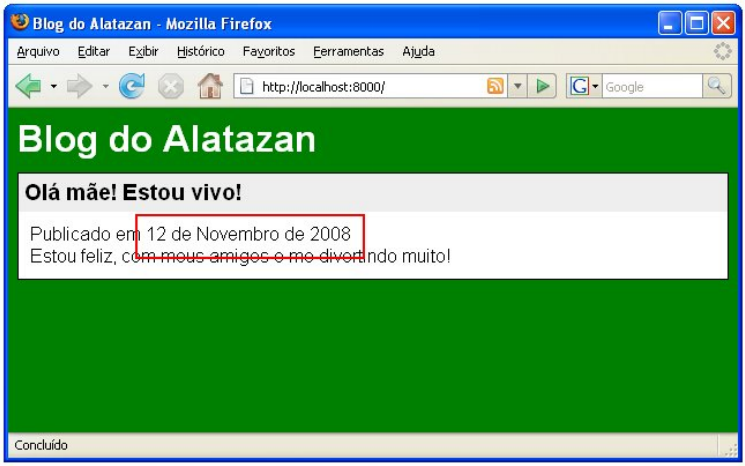
```
LANGUAGE_CODE = 'en-us'
```

E modifique, para ficar assim:

```
LANGUAGE_CODE = 'pt-br'
```

Como deve ter notado, a mudança foi de "**en**glish-**u**nited**s**tates" para "**p**ortuguese-**br**azil".

Salve o arquivo. Feche o arquivo. Volte ao navegador, pressione a tecla **F5** e:



Pronto! Agora estamos falando a mesma língua!

Nota: dê uma olhadinha na interface de administração (http://localhost:8000/admin/) para ver que agora tudo está em bom português brasileiro.

Mas ainda assim, aquela data ficou ruim, pois não possui horas, e ficou um tanto burocrática. Vamos fazer algo mais bacana?

Voltando ao arquivo do template (**artigo_archive.html**) substitua o trecho **|date:"d \d\e F \d\e Y"** por **|timesince**, salve o arquivo e veja como fica:

O problema com arquivos que não são UTF-8

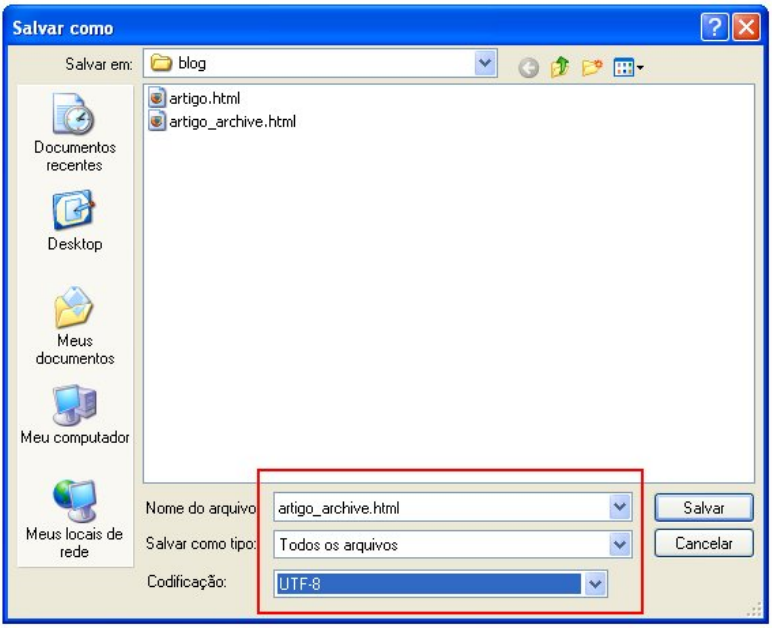
Ahh, agora está muito mais humano - em Katara isso é chamado de "óbvio", mas enfim, não estamos em Katara.

Mas a linguagem ficou estranha, não? "Publicado em 2 dias, 5 horas" não é assim "tão" amigável. Pois então, no arquivo do template, substitua o **em** por **há**.

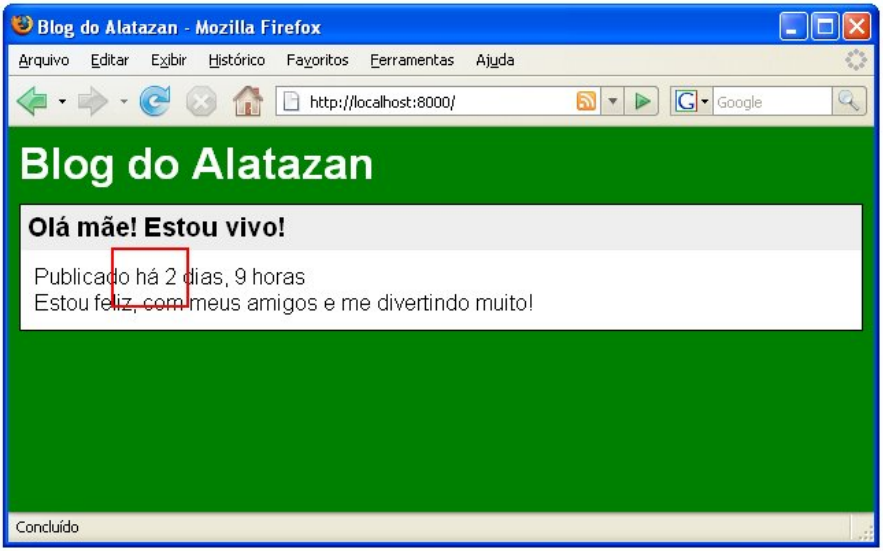
Salve o arquivo. Porém, é possível que você tenha um problema agora, veja:

Se este mesmo erro for exibido para você, é porque você usou um editor que não salva em formato **Unicode** por padrão. Como é o caso do **Bloco de Notas**.

Dessa forma, você deve ir até o menu **Arquivo -> Salvar como** e deixar o mesmo nome do arquivo, mudando os dois campos abaixo do nome para **"Todos os arquivos"** e **"UTF-8"**, respectivamente, como está na imagem abaixo:



Agora, ao retornar ao navegador, veja o resultado:



Pronto. Questão dos caracteres especiais resolvida. Da próxima vez que isso ocorrer, você já sabe como proceder.

Agora observe bem o arquivo de template, e você pode notar o seguinte trecho de código:

```
{% for artigo in latest %}
  <div class="artigo">
    <h2>{{ artigo.titulo }}</h2>


    <div class="conteudo">
      Publicado há {{ artigo.publicacao }}<br/>
      {{ artigo.conteudo }}
    </div>
  </div>
{% endfor %}
```

Veja que o trecho é iniciado com **{% for artigo in latest %}** e finalizado com **{% endfor %}**.

Trata-se de um **template tag**.

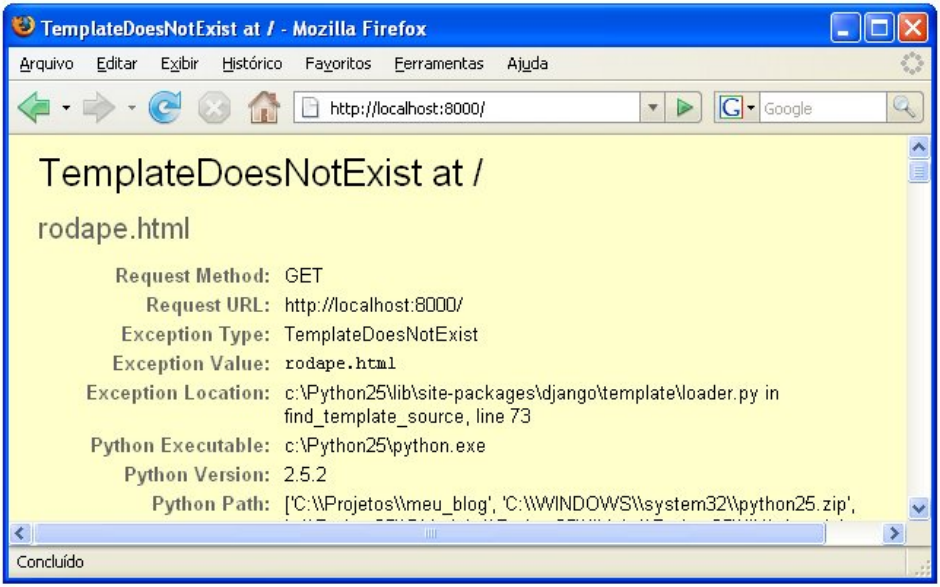
Todos os **Template Tags** iniciam e terminam com **{%** e **%}** respectivamente, e são mais poderosos que qualquer outro elemento em um template, pois é possível trabalhar com blocos, como é o caso desse **laço de repetição** que é aberto pelo template tag **{% for %}** e fechado com seu template tag de fechamento **{% endfor %}**.

Incluindo um template dentro de outro

Após aquele trecho de código citado acima, acrescente este outro trecho de código:

```
{% include "rodape.html" %}
```

Salve o arquivo. Feche o arquivo. Vá ao navegador e atualize com **F5**. Veja o resultado:



Isso acontece, pois a template tag **{% include %}** inclui um template dentro de outro. Maneiro, não é? E como esse outro template incluído não existe, ele exibiu a mensagem de erro.

Vamos então criar esse novo arquivo. Mas pense comigo: **o rodapé de um template não deve ser restrito ao blog, mas deve ser do site como um todo**, concorda? Então dessa vez nós não vamos adicionar esse template à pasta de templates da aplicação **blog**.

Na pasta do projeto, crie uma nova pasta, chamada **"templates"** e dentro dela, crie o arquivo **"rodape.html"** com o seguinte código dentro:

```
<div class="rodape">

Por favor não faça contato. Obrigado pela atenção.

</div>
```

Salve o arquivo. Feche o arquivo.

Criando uma pasta geral para templates do projeto

Mas só isso não é suficiente, pois precisamos dizer ao projeto que ele agora possui uma pasta de templates própria, o que não é feito automaticamente como é feito para aplicações. Portanto, na pasta do projeto, abra o arquivo **settings.py** para edição e localize o seguinte trecho:

```
TEMPLATE_DIRS = (
```

```
    # Put strings here, like "/home/html/django_templates" or
    # "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

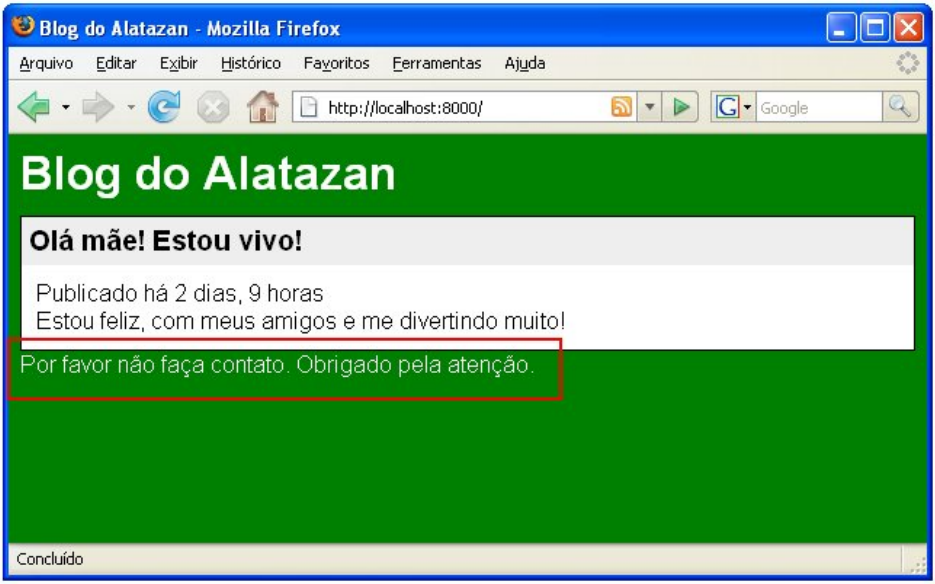
A tupla **TEMPLATE_DIRS** indica as pastas onde podem ser encontrados templates para o projeto, além daquelas que já são automaticamente localizadas dentro das aplicações. O Django não incentiva o uso exagerado dessas pastas "extra" de templates, pois entendemos que a maior partes deles deve ser concentrada em suas devidas aplicações, mas em muitas vezes é inevitável possuir alguns arquivos-chave numa dessas pastas.

Modifique o trecho de código para ficar assim:

```
TEMPLATE_DIRS = (
    'templates',
)
```

Salve o arquivo.

Agora, ao atualizar seu navegador, o resultado será este:



No entanto, há uma coisa desagradável aqui. O caminho **'templates'** não é um caminho absoluto, e isso pode te levar a ter diversos problemas. Vamos portanto

fazer uma boa prática já! No início do arquivo **settings.py**, acrescente as seguintes linhas:

```
import os

PROJECT_ROOT_PATH = os.path.dirname(os.path.abspath(__file__))
```

Agora, modifique a tupla **TEMPLATE_DIRS** para ficar assim:

```
TEMPLATE_DIRS = (

    os.path.join(PROJECT_ROOT_PATH,'templates'),

)
```

E não somente isso. Localize esta linha:

```
DATABASE_NAME = 'meu_blog.db'             # Or path to database
file if using sqlite3.
```

E modifique para ficar assim:

```
DATABASE_NAME = os.path.join(PROJECT_ROOT_PATH,'meu_blog.db')
```

Isso faz com que esses caminhos de pastas e arquivos sejam caminhos completos, e não somente o seu único nome simples.

A template tag {% url %}

Pois agora vamos voltar ao template **"artigo_archive.html"** e localizar a seguinte linha:

```
<h2>{{ artigo.titulo }}</h2>
```

Modifique para ficar assim:

```
<a href="{

      { artigo.get_absolute_url }

      }">

<h2>{

      { artigo.titulo }

      }</h2>

      </a>
```

Isso faz com que o título do artigo passe a ser um link para sua própria página. Mas há ainda uma forma melhor de indicar um link, usando a template tag **{% url %}**.

Essa template tag é bacana, pois é flexível às mudanças. Usando a template tag **{% url %}** para indicar um link, você pode modificar o caminho da URL no arquivo **urls.py** e não precisa se preocupar com os vários links que existem espalhados pelos templates indicando àquela URL, pois eles são ajustados

automaticamente. Portanto, modifique o mesmo trecho de código para ficar assim:

```
<a href="
      {% url blog.views.artigo artigo_id=artigo.id %}">
      <h2>{{ artigo.titulo }}</h2>
</a>
```

Observando o que escrevemos, você vai notar que a template tag indica primeiramente o caminho da view (aplicação **blog**, arquivo **views.py**, view **artigo**) e o argumento **artigo_id** recebendo o código do artigo (**artigo.id**).

Verificando a URL em questão, veja se reconhece as semelhanças:

```
(r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
```

Notou o fator de ligação? Observe os elementos **blog.views.artigo** e **artigo_id**.

Salve o arquivo. Feche o arquivo. E agora, ao carregar a página no navegador novamente, você vai notar o link para a página no artigo.

A herança de templates

Pois bem, falando na página do artigo, vamos ver como ela está?

Localize em seu navegador a seguinte URL:

```
http://localhost:8000/artigo/1/
```

Como pode ver, a mudança visual que fizemos no template da página inicial não afetou em nada a página do artigo. Isso ocorre porque de fato os dois templates são distintos. Mas existe uma solução pra isso!

No conceito de **Programação Orientada a Objetos** há dois elementos importantes, e algumas vezes conflitantes, chamados **herança** e **composição**. No sistema de templates do Django, você pode trabalhar com ambos os conceitos, adotando o mais adequado para cada situação.

Nós já trabalhamos o conceito de **composição** lá atrás, quando você conheceu a template tag **{% include %}**, que trabalha na idéia de compôr um template acrescentando outros templates de fim específico (como apresentar tão somente um rodapé, por exemplo) como parte dele.

Agora então vamos fazer uso da **herança**.

O conceito de herança trata-se de concentrar uma parte **generalizada** em um template que será **herdado** por outros, ou seja, estes outros serão uma cópia daquele, modificando as partes que assim o desejarem. Isso é **muito útil** para se evitar re-trabalho na construção de layouts de páginas.

Pois então vamos ao trabalho.

Na pasta **"templates"** do **projeto** (**meu_blog/templates**), crie um novo arquivo, chamado **"base.html"**, e escreva o seguinte código dentro:

```
<html>
<head>
<title>
  {% block titulo %}Blog do Alatazan{% endblock %}
</title>
<link
   rel="alternate"
   type="application/rss+xml"
   title="Ultimos artigos do Alatazan"
   href="/rss/ultimos/"
/>
<style type="text/css">
body {
  font-family: arial;
  background-color: green;
  color: white;
}

h1 {
  margin-bottom: 10px;
}

h2 {
  margin: 0;
  background-color: #eee;
  padding: 5px;
  font-size: 1.2em;
}

.artigo {
  border: 1px solid black;
```

```
  background-color: white;
  color: black;
}

.conteudo {
  padding: 10px;
}
</style>
</head>

<body>

<h1>{% block h1 %}Blog do Alatazan{% endblock %}</h1>

{% block conteudo %}{% endblock %}

{% include "rodape.html" %}

</body>
</html>
```

Você pode notar que se trata do mesmo código HTML que fizemos no template **"artigo_archive.html"**, com algumas modificações. E essas modificações foram a remoção de partes do código que eram específicas daquela página (uma listagem de artigos), e a criação de áreas potencialmente modificáveis. Essas áreas são as template tags **{% block %}**.

A template tag **{% block %}** indica que naquele espaço aberto por **{% block %}** e fechado por **{% endblock %}**, os templates que herdarem este **podem fazer mudanças** adequadas à sua realidade.

Salve o arquivo. Vamos agora voltar a editar o template **"artigo_archive.html"**, da pasta **blog/templates/blog** e modificar para todo o arquivo ficar assim:

```
{% extends "base.html" %}


{% block conteudo %}
{% for artigo in latest %}
  <div class="artigo">
```

```
    <a href="
      {% url blog.views.artigo artigo_id=artigo.id %}">
      <h2>{{ artigo.titulo }}
      </h2>
    </a>

    <div class="conteudo">
      Publicado há {{ artigo.publicacao|timesince }}<br/>
      {{ artigo.conteudo }}
    </div>
  </div>
{% endfor %}
{% endblock %}
```

Puxa! Diminuiu bastante, não foi?

Sim, e o que fizemos foi exatamente o oposto do template **base.html**: removemos as partes generalizadas e levadas para o template herdado e deixamos somente as partes específicas desse template.

Note a template tag **{% extends "base.html" %}** no início do template. É ela quem diz que este template será uma **extensão** daquele que criamos (**base.html**). Essa template tag deve sempre estar posicionada no início do template.

A partir do momento em que um template extende outro, **qualquer código que estiver fora** de uma template tag **{% block %}** será ignorado. Portanto, veja que o trecho de código específico desse template foi colocado dentro do bloco **{% block conteudo %}**.

Os demais blocos não foram declarados ou citados porque nesta página não precisamos modificá-los.

Salve o arquivo. Feche o arquivo. E ao atualizar o navegador com **F5**, verifique que o visual da página inicial permanece o mesmo.

Agora vamos fazer o mesmo trabalho no template **artigo.html** da pasta **blog/templates/blog**. Abra esse arquivo para edição e modifique para ficar assim:

```
{% extends "base.html" %}

{% block titulo %}{{ artigo.titulo }} -
{{ block.super }}{% endblock %}
```

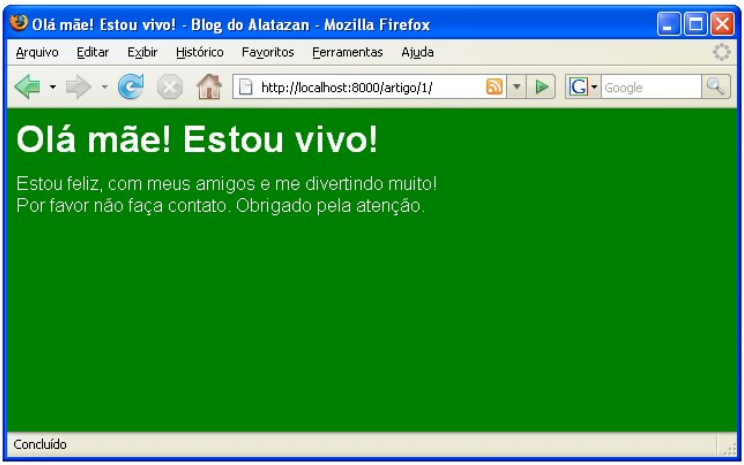
```
{% block h1 %}{{ artigo.titulo }}{% endblock %}


{% block conteudo %}
{{ artigo.conteudo }}
{% endblock %}
```

Veja que extendemos o mesmo template de herança (**base.html**). Mas desta vez declaramos os outros blocos, pois precisávamos mudar seus conteúdos.

O caso do bloco **{% block titulo %}** é especial, pois ele possui um elemento bastante útil aqui, o **{{ block.super }}**.

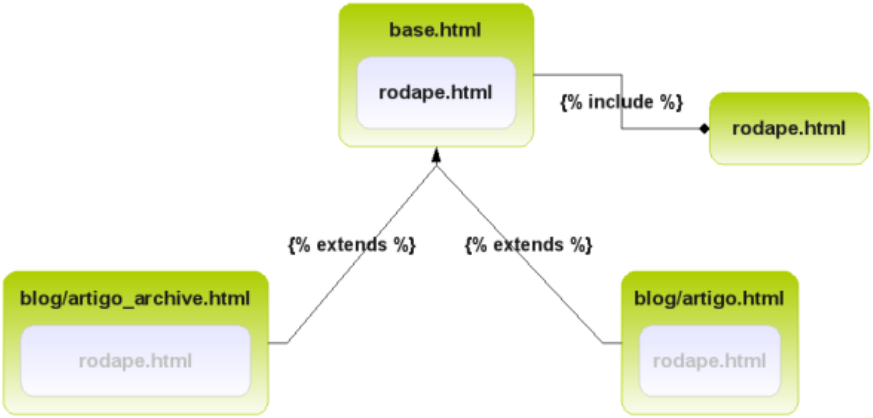
O **{{ block.super }}** é uma variável que carrega o conteúdo do bloco no template herdado, e quando ele é informado, isso significa que naquele lugar deve ser mantido o conteúdo do bloco original. Ou seja, nós mudamos o **título** da página, mas mantivemos o original, apenas acrescentando à esquerda o trecho adicional que queríamos.

Agora salve o arquivo. Feche o arquivo. Vá até o navegador e veja o resultado:



Gostou do resultado?

Agora podemos representar a herança e composição dos nossos templates da seguinte forma:



Agora, partindo para separar as coisas...

- Se eu gostei? Mas claro, isso é fascinante, logo logo vou poder colocar meu blog no ar e...

Num rápido relance, Alatazan interrompeu seu discurso ao ver um vulto passar pela janela. Ele não identificou aquele movimento rápido, mas algo lhe lembrou daquela vez que havia conhecido Ballzer, na praça próxima de sua casa, em Katara.

Ele coçou a cabeça e percebeu que já era tarde. A empolgação o manteve ligado nas explicações de Cartola e Nena e agora era a hora de voltar rapidamente para casa. De outra forma, poderia perder a aula de xadrez daquele dia.

- Alatazan, leve esse livro com você, vai te ajudar a pegar alguns lances mais comuns de Xadrez - Nena pegou um livro na estante e entregou a ele. - você vai notar que cada peça tem uma dinâmica diferente, e algumas delas nem se pode chamar de dinâmica, de tão limitadas que são... mas cada uma cumpre o seu papel.

Alatazan agradeceu com um gesto típico de Katara e se despediu.

No próximo capítulo vamos separar algumas coisas do template, aplicar imagens e conhecer um pouco mais sobre como fazer isso.

Capítulo 8: Trabalhando com arquivos estáticos

O que sensibiliza Alatazan desde que chegou à Terra é que nós humanos possuímos uma virtude que também é importante em Katara. Seres orgânicos possuem **individualidade**.

•Cada flor, mesmo sendo da mesma espécie e da mesma planta, possui um tamanho diferente, um tom de cor particular. - dizia ele consigo mesmo enquanto tomava um sorvete na praça.

O sorvete caiu no chão e balançou a cabeça. Mas continuou:

• Cada pessoa tem um jeito diferente de agir, uma importância particular para o todo, e tratar a todos como se fossem iguais seria desperdiçar o que cada um tem de melhor...

Agora um ou dois pombos mexiam no sorvete espalhado pelo chão.

Filosofias à parte, Alatazan se levantou e seguiu para a casa de Cartola.

• Vamos lá meu amigo, hoje é dia de estudar outra peça importante na engrenagem do site

Os arquivos estáticos

Um site não se resume a páginas em HTML. Ele também possui imagens, arquivos CSS, JavaScript, Flash e outros formatos diversos.

Mas há uma diferença fundamental entre umas e outras partes do site, e isso não é restrito ao tipo de conteúdo que será enviado para o navegador, e sim a como aquele conteúdo se comporta no site.

Ele é **dinâmico** ou **estático**?

A página inicial do blog apresenta um resultado diferente a cada vez que é criado um novo artigo. Seu conteúdo poderia variar também dependendo do usuário

que está autenticado naquela hora. Essa é uma página **dinâmica**.

Já a foto sorridente de Alatazan no canto superior esquerdo do site, será a mesma até que alguém substitua o arquivo por outro. Então, ela é **estática**.

Mas que foto? Não há nenhuma foto por lá. Então é isso que vamos fazer.

Antes de mais nada, não se esqueça de executar o projeto no Django, clicando duas vezes sobre o arquivo **executar.bat** da pasta do projeto.

Agora abra o arquivo de template **artigo_archive.html, da pasta **blog/templates/blog** para edição, e localize a seguinte linha:

```
{% block conteudo %}
```

Agora acrescente a seguinte linha abaixo dela:

```
<img src="{{ MEDIA_URL }}foto.jpg"/>
```

Salve o arquivo. Feche o arquivo. Essa tag HTML vai exibir uma imagem chamada **foto.jpg** no caminho apontado pela varíavel **{{ MEDIA_URL }}**.

A variável **{{ MEDIA_URL }}** contém o mesmo valor contido na **setting MEDIA_URL**.

Então vamos agora abrir o arquivo **setings.py** da pasta do projeto para edição, e localizar esta setting:

```
MEDIA_URL = ''
```

A setting **MEDIA_URL** tem a função de indicar o endereço que contém **arquivos estáticos** do projeto.

Isso não indica que seja possível ter apenas um. Em grandes projetos é necessário haver mais caminhos para arquivos estáticos, então são tomados outras práticas para isso. Mas sempre que possível, é recomendável ter apenas um lugar que forneça esses arquivos. E o caminho mais indicado é o **/media/**. Portanto, modifique essa linha para ficar assim:

```
MEDIA_URL = '/media/'
```

Mas só isso não é suficiente. Localize a setting **MEDIA_ROOT** e veja como está:

```
MEDIA_ROOT = ''
```

Esta setting indica o caminho no HD onde estão os arquivos estáticos. Modifique esta setting para ficar assim:

```
MEDIA_ROOT = os.path.join(PROJECT_ROOT_PATH, 'media')
```

Você deve ser lembrar de que já usamos essa função **os.path.join()** com a setting **PROJECT_ROOT_PATH** não é mesmo? E ela retorna para a setting **MEDIA_ROOT** o caminho completo para a pasta do projeto (**meu_blog**) somada

com a subpasta **media**.

E por fim, ainda temos uma coisa para fazer. Localize a setting **ADMIN_MEDIA_PREFIX** e veja como está:

```
ADMIN_MEDIA_PREFIX = '/media/'
```

Esta setting indica o caminho para os arquivos estáticos do **Admin** - a interface automática de administração do Django. Acontece que ela está apontando exatamente para o mesmo caminho que a setting **MEDIA_URL** e isso vai dar em **conflito**. Então a modifique para ficar assim:

```
ADMIN_MEDIA_PREFIX = '/admin_media/'
```

Você deve estar se perguntando **"mas se estas são as melhores práticas, então porquê essas settings já não vêm dessa forma?"**. Ahn... **melhores práticas** é uma expressão muito forte. E este assunto trata de variações diversas em projetos distintos. Digamos que estas sejam as melhores práticas para sites pequenos ou sites em desenvolvimento. Mas ainda assim não são as únicas boas práticas para isso. Portanto, o Django tenta ser o mais discreto possível, deixando essa decisão para você.

Salve o arquivo. Feche o arquivo.

Acha que é só isso? Não é. O caminho que indicamos à setting **MEDIA_URL** não existe no arquivo **urls.py** e como este é um **ambiente de desenvolvimento**, temos isso a fazer ali: abrir o arquivo **urls.py** para edição e adicionar a seguinte URL:

```
    (r'^media/(.*)$', 'django.views.static.serve',
     {'document_root': settings.MEDIA_ROOT}),
```

Notou que fizemos a ligação entre as settings **MEDIA_URL** e **MEDIA_ROOT**? A segunda dá suporte à primeira.

Mas o elemento **settings.MEDIA_ROOT** ali não vai funcionar, pois é preciso dizer quem é o **"settings"** em questão. Portanto, localize a seguinte linha no arquivo:

```
from django.conf.urls.defaults import *
```

E adicione abaixo dela:

```
from django.conf import settings
```

Agora o arquivo **urls.py** ficou assim:

```
from django.conf.urls.defaults import *
from django.conf import settings


# Uncomment the next two lines to enable the admin:
```

```
from django.contrib import admin
admin.autodiscover()

from blog.models import Artigo
from blog.feeds import UltimosArtigos

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
         'date_field': 'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
    (r'^media/(.*)$', 'django.views.static.serve',
     {'document_root': settings.MEDIA_ROOT}),
)
```

Salve o arquivo. Feche o arquivo.

Pronto, já temos a URL funcionando. Mas agora precisamos ter uma pasta **"media"** na pasta do projeto, com o arquivo **foto.jpg** dentro, certo?

Então vá até a pasta do projeto e crie a pasta **media**.

A foto do Alatazan é esta:

Você pode usá-la, baixando da seguinte URL:

```
http://www.aprendendodjango.com/gallery/foto-de-alatazan/file/
```

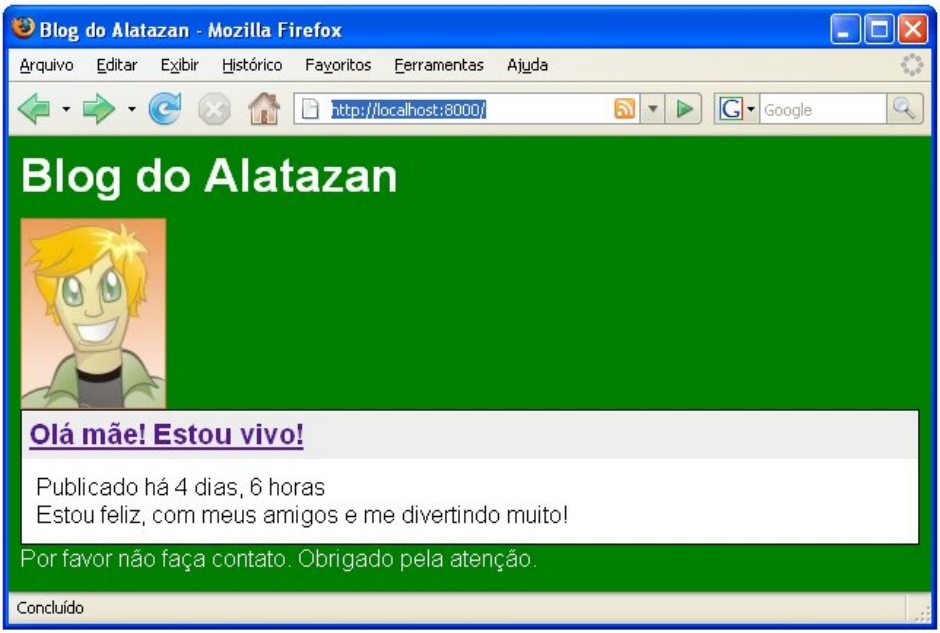
Ou pode usar uma foto sua. Como preferir. O que importa é que essa imagem deve ser do formato **JPEG**, o nome do arquivo deve ser **"foto.jpg"**, e deve ser

colocada na pasta **media**. Isso porque este é o endereço indicado pela tag HTML desta imagem, lá no template.

Feito isso, agora temos a imagem da foto do Alatazan, devidamente salva na pasta correta e com as todas referências finalizadas. Vamos ao navegador? Pois bem, localize no seu navegador a URL principal do blog:

```
http://localhost:8000/
```

E veja o resultado:



Bacana! Agora temos uma imagem posicionada em nosso layout!

É possível que haja uma dúvida em sua cabeça agora: **"mas o Django só suporta imagens JPEG?"**

Não. isso não está sob poder do Django. O Django suporta qualquer imagem ou outro formato de arquivo estático, seja ele Shockwave Flash (.swf), imagens (.gif, .jpg, .png e outras), mídia digital (.mp3, .wma, .ogg e outras), JavaScript (.js), CSS (.css) ou qualquer outro formato de arquivo **estático**. Essa é uma definição que não depende do Django, mas sim de seu servidor e navegador que vão trabalhar com esses arquivos.

Agora vamos fazer outra modificação: separar o **CSS** do template, e criar um arquivo estático para conter somente o CSS. Vamos?

Na pasta do projeto, abra a pasta **templates**. Nesta pasta, abra o arquivo **base.html** para edição, e substitua o seguinte trecho de código:

```
<style type="text/css">
body {
  font-family: arial;
  background-color: green;
  color: white;
}

h1 {
  margin-bottom: 10px;
}

h2 {
  margin: 0;
  background-color: #eee;
  padding: 5px;
  font-size: 1.2em;
}

.artigo {
  border: 1px solid black;
  background-color: white;
  color: black;
}

.conteudo {
  padding: 10px;
}
</style>
```

Por este:

```
<link rel="stylesheet" type="text/css"
href="{{ MEDIA_URL }}layout.css"/>
```

Agora o nosso template base.html ficou assim:

```
<html>
<head>
<title>
{% block titulo %}Blog do Alatazan{% endblock %}
</title>
<link
rel="alternate"
type="application/rss+xml"
title="Ultimos artigos do Alatazan"
href="/rss/ultimos/"
/>
<link rel="stylesheet" type="text/css"
href="{{ MEDIA_URL }}layout.css"/>
</head>

<body>

<h1>{% block h1 %}Blog do Alatazan{% endblock %}</h1>

{% block conteudo %}{% endblock %}

{% include "rodape.html" %}

</body>
</html>
```

Salve o arquivo. Feche o arquivo. E veja como ficou:

Opa! Parece que perdemos o nosso visual verde bandeira! Sim, mas vamos agora criar o arquivo de CSS para voltar ao normal.

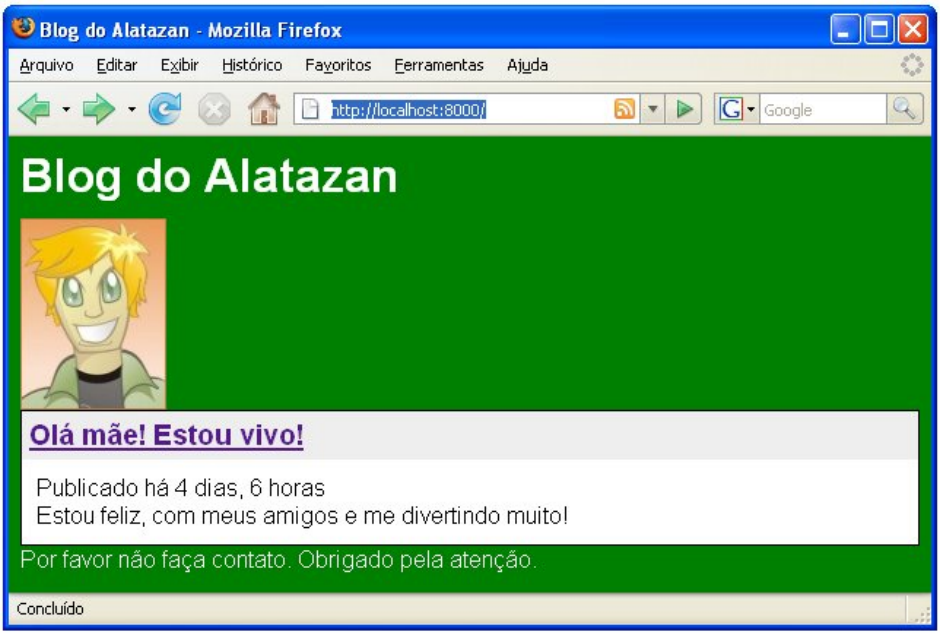
Na pasta **"media"** do projeto, crie um arquivo chamado **"layout.css"** e escreva o seguinte código dentro:

```
body {
  font-family: arial;
  background-color: green;
  color: white;
}

h1 {
  margin-bottom: 10px;
}

h2 {
  margin: 0;
  background-color: #eee;
  padding: 5px;
```

```
  font-size: 1.2em;
}

.artigo {
  border: 1px solid black;
  background-color: white;
  color: black;
}

.conteudo {
  padding: 10px;
}
```

Notou que é exatamente o mesmo que removemos do template **base.html**, exceto pela ausência das tags **<style>**?

Pois agora salve o arquivo. Feche o arquivo. Volte ao navegador e veja que o visual do site voltou ao normal:



Agora, vá novamente à página do artigo, no navegador, na seguinte URL:

```
http://localhost:8000/artigo/1/
```

O resultado será o seguinte:

Opa! Onde foi parar o estilo CSS desta página?

Acontece que esta página vem que uma **view** que foi feita manualmente por você, e esta não possui um **contexto** que contenha um valor para a variável **{{ MEDIA_URL }}**, e isso fez com que sua referência ficasse vazia, perdendo seu estilo.

Mais pra frente vamos ver isso em detalhes, mas agora, vamos nos atentar e resolver o problema.

Na pasta da aplicação **"blog"**, abra o arquivo **views.py** para edição, e modifique o arquivo para ficar da seguinte forma:

```
from django.shortcuts import render_to_response
from django.template import RequestContext


from models import Artigo


def artigo(request, artigo_id):
    artigo = Artigo.objects.get(id=artigo_id)
    return render_to_response('blog/artigo.html', locals(),
        context_instance=RequestContext(request))
```

O que nós fizemos? Nós acrescentamos a seguinte linha:

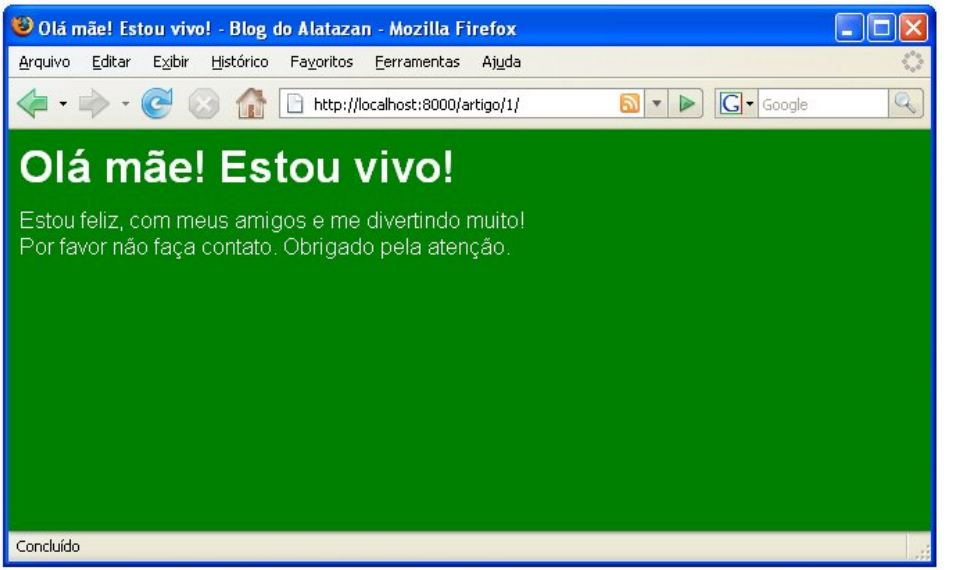
```
from django.template import RequestContext
```

E modificamos esta outra:

```
    return render_to_response('blog/artigo.html', locals(),
        context_instance=RequestContext(request))
```

Essas modificações fazem com que algumas variáveis especiais (incluindo a **{{ MEDIA_URL }}**) sejam válidas nos templates.

Salve o arquivo. Feche o arquivo. Vá ao navegador, atualize com **F5** e veja o resultado:



Pronto! Agora sim, o problema foi resolvido.

De um extremo ao outro: do estático para as páginas planas!

O tempo passou voando e Alatazan já estava curioso em conhecer ferramentas para a criação de imagens: o **Inkscape** e o **Gimp**. Essas duas ferramentas são fantásticas para a criação de imagens para sites. Nena já havia comentado sobre elas.

Mas ele não quis perder o foco, e comentou com o dono da casa:

- Bom, hoje foi bem diferente...
- Mas é uma parte muito, muito importante do site, Alatazan. Não existem sites sem CSS e imagens, ou pelo menos não existem **bons** sites sem eles -

interrompeu Nena, muito enfática - e este é um ponto bem confuso, às vezes...

• Sim, é mesmo... é um pouco confuso. Mas depois que você aprende, é simples entender que:

  • É preciso ajustar as settings **MEDIA_URL**, **MEDIA_ROOT** e **ADMIN_MEDIA_PREFIX**

  • A setting **MEDIA_URL** é uma URL para a pasta **MEDIA_ROOT**

  • É preciso criar a pasta **media** na pasta do projeto

  • É preciso criar a URL **^media/**

  • Toda referência a arquivos estáticos nos templates devem iniciar com **{{ MEDIA_URL }}**

• Muito bom, muito bom! - foi a vez de Cartola.

• Agora que tenho a minha foto na página principal, eu estou sentindo falta de um lugar para falar quem sou eu, e alguma informação a mais...

• Sim. Vamos fazer assim: amanhã vamos estudar sobre **Páginas Planas**, é pra isso que elas servem! Agora vamos embora que o calor está nos *convidando* pra um passeio lá na rua.

Capítulo 9: Páginas de conteúdo são FlatPages

Folheando o livro de Xadrez que Nena lhe emprestou, Alatazan notou aquelas seções de informações do livro, como uma breve biografia do autor, na contracapa, e a página onde haviam seus agradecimentos. Na última página do livro havia também um texto promocional de outros livros do mesmo autor, sua fotografia e um cartão da loja onde o livro havia sido comprado.

É comum haver esse tipo de páginas na maior parte dos sites.

Isso é porquê você precisa dizer quem é, do que se trata e outros assuntos relacionados para dar solidez e demonstrar transparência ao seu usuário. Seu legado é importante.

Como uma luva: as Páginas Planas

Para páginas de conteúdo simples assim, as **FlatPages** - ou traduzindo - as **"Páginas Planas"**, encaixam-se como luvas, pois não é necessário saber programar em Django para criá-las, basta que o programador tenham habilitado esse recurso e criado um template para isso e todo o resto pode ser feito pelo **Admin** - a interface de administração do Django.

Vamos logo colocar a mão na massa, pois **não é preciso muito esforço** para chegar ao final da lição de hoje.

Antes de tudo, inicie seu projeto clicando duas vezes no arquivo **executar.bat**.

Agora abra a pasta do projeto, e edite o arquivo **settings.py** para adicionar a aplicação **flatpages** ao projeto e seu middleware.

Faça assim: vá até o final do arquivo, na setting **INSTALLED_APPS** e adicione a seguinte linha:

```
'django.contrib.flatpages',
```

Com isso, o trecho de código dessa setting ficou assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',

    'blog',
)
```

Agora vamos adicionar **o middleware que faz todo o trabalho mágico** das FlatPages. Localize a setting **MIDDLEWARE_CLASSES** ainda no arquivo **settings.py** e adicione a seguinte linha:

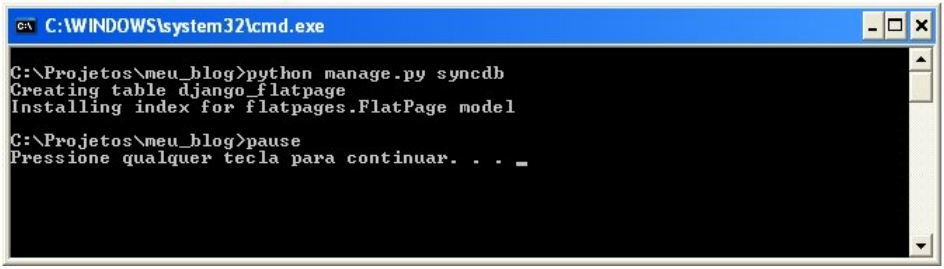
```
'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
```

Agora o trecho de código da setting **MIDDLEWARE_CLASSES** ficou assim:

```
MIDDLEWARE_CLASSES = (
'django.middleware.common.CommonMiddleware',
'django.contrib.sessions.middleware.SessionMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
)
```

Salve o arquivo. Feche o arquivo.

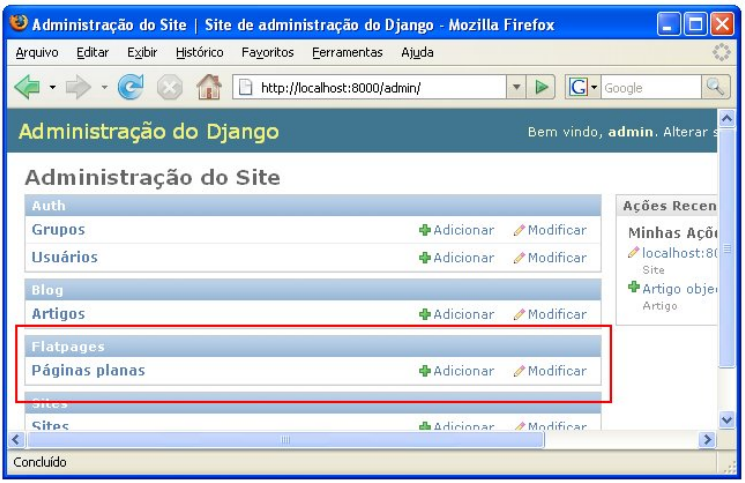
Agora vamos gerar o banco de dados para ver o efeito que isso causou em nosso projeto. Na pasta do projeto, clique duas vezes sobre o arquivo **gerar_banco_de_dados.bat** que deve executar da seguinte forma:



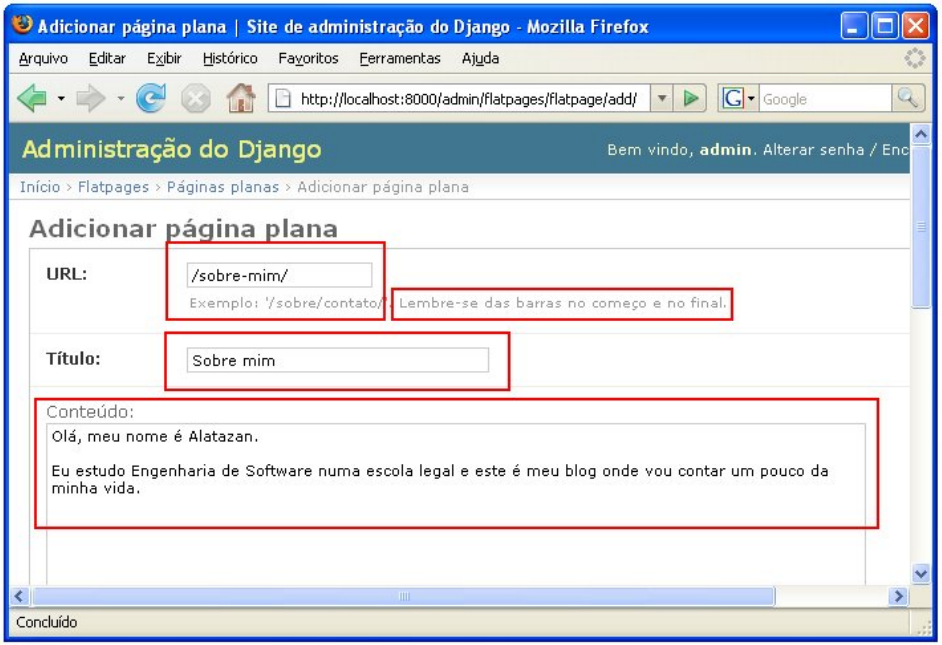
Com o banco de dados gerado, podemos abrir o navegador na URL do admin:

```
http://localhost:8000/admin/
```

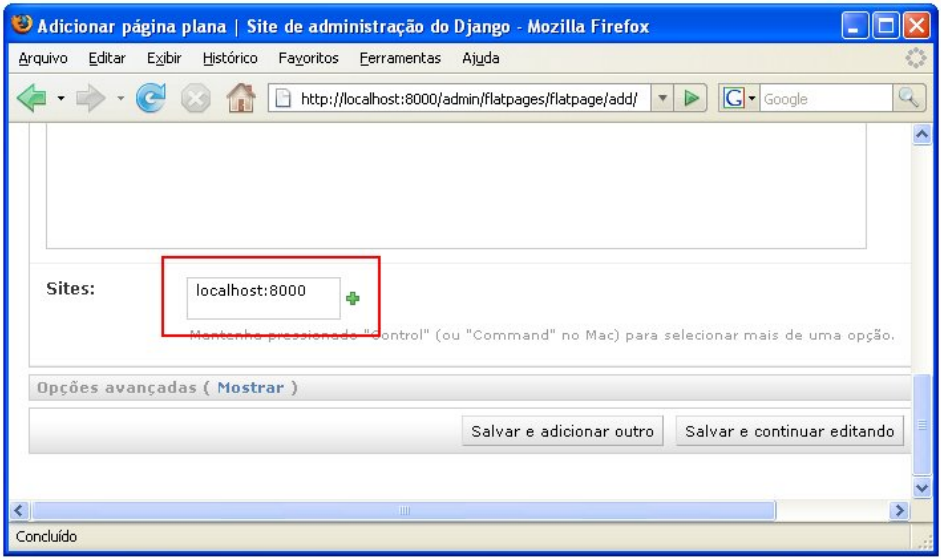
E temos agora uma nova seção para **Páginas Planas**:



Clique no link **"Adicionar"** à direita da palavra **"Páginas Planas"** e adicione uma nova URL da seguinte forma:



... e ao rolar a **barra de rolagem**, ainda há este outro campo para informar:



Ou seja, informe os seguintes valores para os respectivos campos:

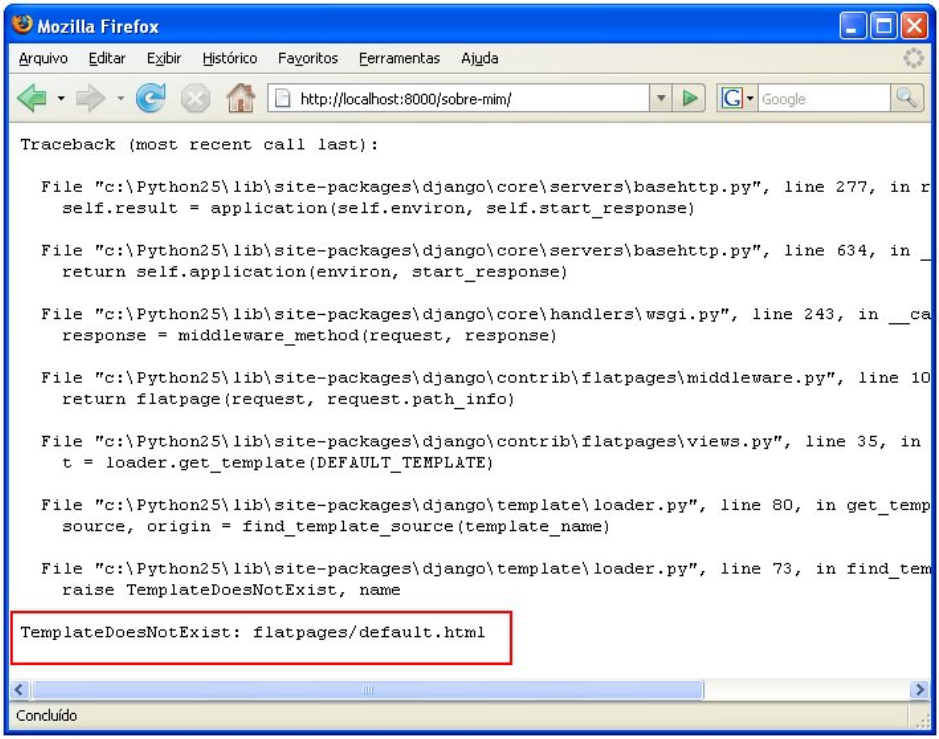
- URL: **/sobre-mim/**
- Título: **Sobre mim**
- Conteúdo: **esteja livre para informar o texto como quiser, mas faça questão que esse texto tenha mais de uma linha**
- Sites: **localhost:8000**

Uma informação importante aqui: **não se esqueça** da barra (/) **no início e no final** do **campo URL**. Isso é importante para que o middleware consiga casar a URL com a FlatPage, ok?

Pois bem, agora clique sobre o botão **"Salvar"** e carregue no navegador o seguinte endereço:

```
http://localhost:8000/sobre-mim/
```

E veja o resultado a seguir:



Opa! Temos um erro aqui!

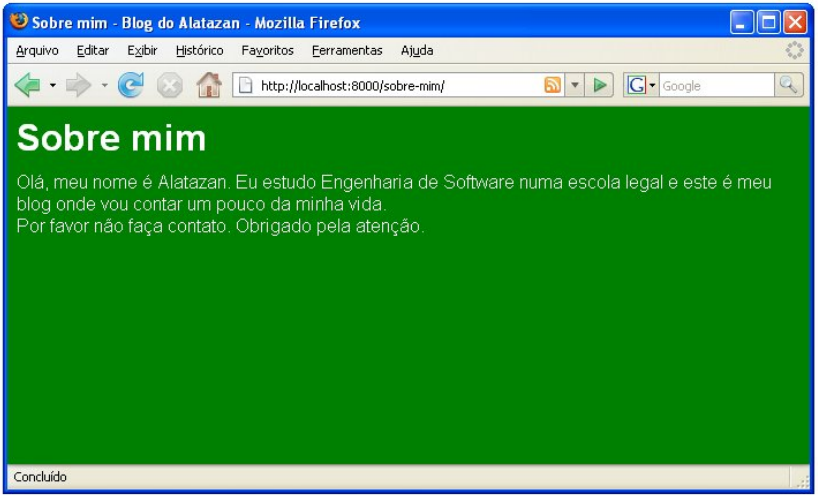
Mas é simples: **para uma FlatPage funcionar, é necessário haver um template onde ela será "encaixada"**.

Portanto, vamos agora à pasta do projeto e abrir dali a pasta **"templates"**. Dentro da pasta **"templates"**, crie uma nova pasta, chamada **"flatpages"**. Dentro da nova pasta, crie o arquivo **"default.html"**. Abra-o para edição e escreva o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}{{ flatpage.title }} -
{{ block.super }}{% endblock %}


{% block h1 %}{{ flatpage.title }}{% endblock %}


{% block conteudo %}
```

```
{{ flatpage.content }}
{% endblock %}
```

Salve o arquivo. Volte ao navegador e pressione a tecla **F5** para atualizar. Veja como ficou:



Difícil? Uma baba! Mas há uma coisa feia ali. Lembra-se de que escrevemos um texto com **duas linhas**? Sim, mas elas se transformaram em uma só... mas o que aconteceu?

A resposta é simples: o HTML gerado para textos com salto de linha ignora esses saltos de linha a menos que haja a tag HTML **
** entre elas. Mas o Django é muito legal, então há uma solução mais amigável para isso.

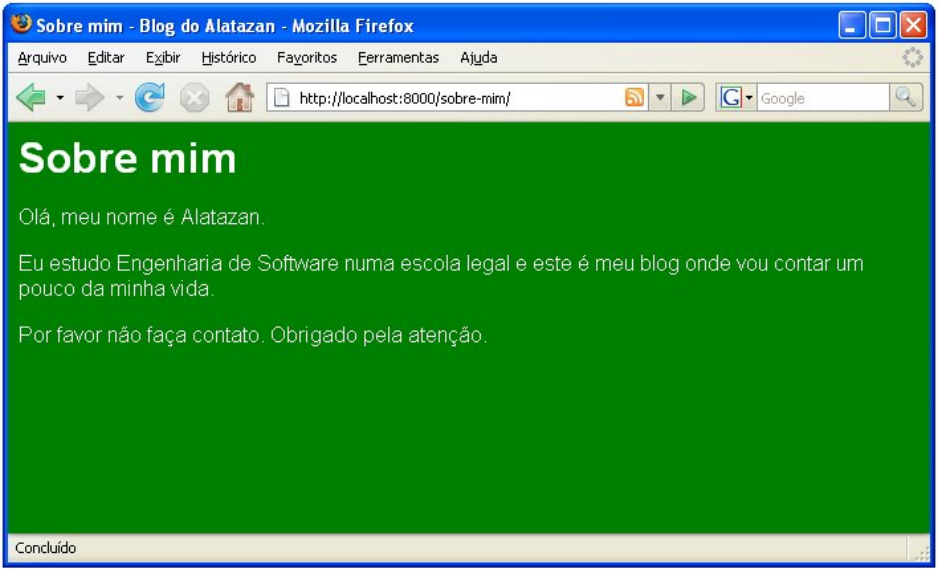
Volte ao arquivo de template que acabamos de criar e localize a seguinte linha:

```
{{ flatpage.content }}
```

Agora a modifique para ficar assim:

```
{{ flatpage.content|linebreaks }}
```

Salve o arquivo. Feche o arquivo. Vá ao navegador, atualize a página e veja como ficou:



Pronto! **FlatPages** são isso que você acabou de ver!

Chega de mau humor, Alatazan, claro que queremos fazer contato com você!

- Tranquilo, tranquilo hein gente... - falou um Alatazan relaxado.

- Tranquilo? Isso é até sem graça de tão tranquilo - foi a resposta empolgada do Cartola

- É, bastou **adicionar a aplicação** ao projeto, **adicionar o middleware** e **criar o template**. Só isso!

- E aí é só ir até o **Admin** e criar quantas páginas quiser, usando HTML e nunca se esquecendo de que a URL da Flatpage **sempre inicia e termina com a barra** (/). - completou Nena.

- Agora tem uma coisa esquisita nesse seu blog, cara. Você deixou um rodapé escrito "Por favor, não faça contato..." não é assim que se faz. Porque você não cria uma **página de Contato**?

Alatazan olhou para ele como se fosse um daqueles índios pelados da baía de Guanabara vendo navios portugueses pela primeira vez.

- É! É isso, vamos fazer uma página de contato, então!

Pois bem, então é isso pessoal!

Capítulo 10: Permitindo contato do outro lado do Universo

•Shhhéeeeght! Pzzt! SShhháptt! Tzóing! Pzzziiitt! Pam-pararampam. Pam! Pam

Lá estava Alatazan em sua nave, uns 500 km acima do chão, já quase no limite da atmosfera, tentando regular o canal para fazer contato com seu planeta.

O sistema de comunicação de sua nave utiliza **dobradura de nível 6** no espaço, com envergadura de **70º a 90º** e isso pode ser seriamente afetado quando há uma tempestade de asteroides nas redondezas. Não faz muito tempo que Katara descobriu o **rádio** como uma simples solução para comunicações a grandes distâncias, e foi também nesta época que eles souberam da existência do Planeta Terra e de nossa Internet.

Nessa hora ficou ainda mais evidente que Alatazan não podia viver isolado em seu espaço, **é preciso ter um meio para receber contato** de outras pessoas, **não importa quão distante elas estejam**.

Então, naquele dia eles criaram:

Uma página para Contato

A página de Contato pode variar de um site para outro, mas não tem erro: no mínimo ela precisa ter quem está fazendo contato e qual é a sua mensagem. Essas informações são transformadas em um texto e enviadas por e-mail do servidor para a sua caixa de mensagens, sem precisar expôr seu e-mail para estranhos.

Aliás, esse era o medo que Alatazan tinha quando escreveu sua nada amigável frase "Por favor não faça contato. Obrigado pela atenção.". Ele não queria expôr seu e-mail para qualquer um.

Antes de mais nada, lembre-se de executar seu projeto, clicando duas vezes sobre o arquivo "executar.bat" da pasta do projeto.

Agora, na pasta "templates" do projeto, abra o arquivo "rodape.html" e o

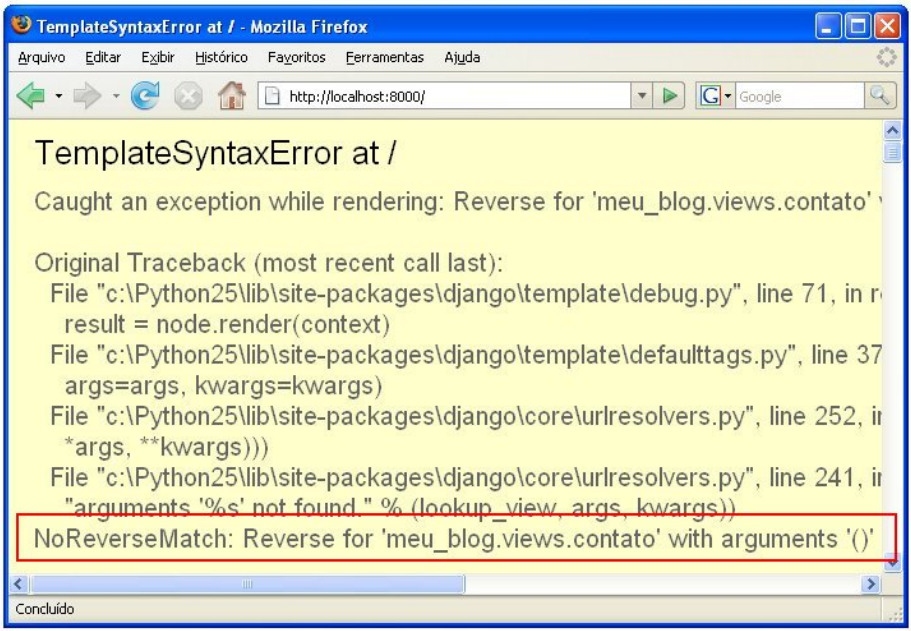
modifique, para ficar assim:

```
<div class="rodape">
<a href="
{% url views.contato %}
">
Para fazer contato comigo, clique aqui.
</a>
</div>
```

Salve o arquivo. Feche o arquivo. Agora vamos ao navegador para ver como fica a página principal do site:

```
http://localhost:8000/
```

Veja:



Puxa vida, mas que tamanho de erro é este? Mas a parte mais importante dessa mensagem de erro é isto:

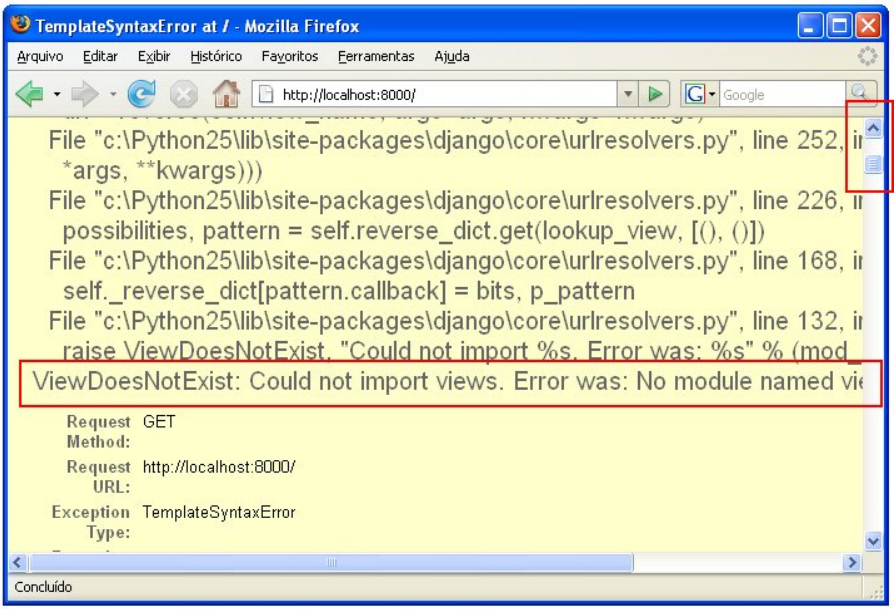
```
NoReverseMatch: Reverse for 'meu_blog.views.contato' with
arguments '()' and keyword arguments '{}' not found.
```

Bastante evidente, certo? Não existe uma URL para a view localizada em *'meu_blog.views.contato', é isso que ele está dizendo.

Vamos criar essa URL então: abra o arquivo "urls.py" da pasta do projeto para edição e insira esta linha ao final da chamada patterns():

```
    (r'^contato/$', 'views.contato'),
```

Salve o arquivo. Feche o arquivo. De volta ao navegador, pressione F5 e olha o resultado:



Bom, continuamos com uma mensagem de erro, mas agora ela mudou, veja:

```
ViewDoesNotExist: Could not import views. Error was: No module
named views
```

Em outras palavras: não é possível importar* o módulo views, pois ele não existe. Pois então vamos criá-lo. Na pasta do projeto (meu_blog), crie um novo arquivo chamadado views.py e escreva o seguinte código dentro:

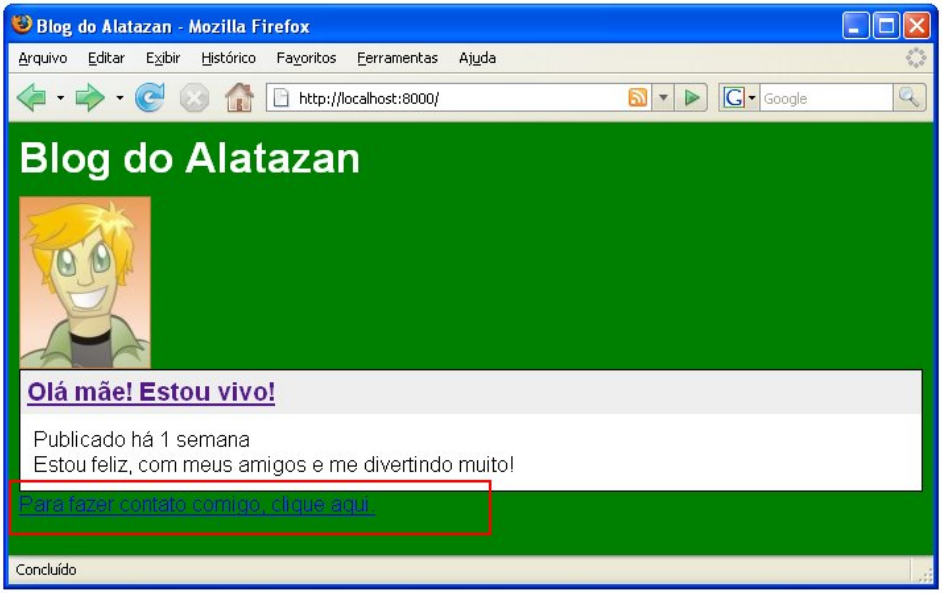
```
from django.shortcuts import render_to_response
from django.template import RequestContext


def contato(request):
    return render_to_response(
        'contato.html',
        locals(),
        context_instance=RequestContext(request),
```

```
)
```

Salve o arquivo. Feche o arquivo.

Você deve ter estranhado que criamos um novo arquivo views.py na pasta do projeto, ao invés de usar o arquivo views.py da pasta da aplicação blog. É que esta não é uma funcionalidade do blog, e sim do site como um todo. Devemos lembrar que o projeto pode ter mais coisas no futuro, além do seu blog. Portanto, um arquivo views.py na pasta do projeto para funções que são específicas do projeto faz bastante sentido.

Mas então, de volta ao navegador, pressione F5 e veja como ficou:



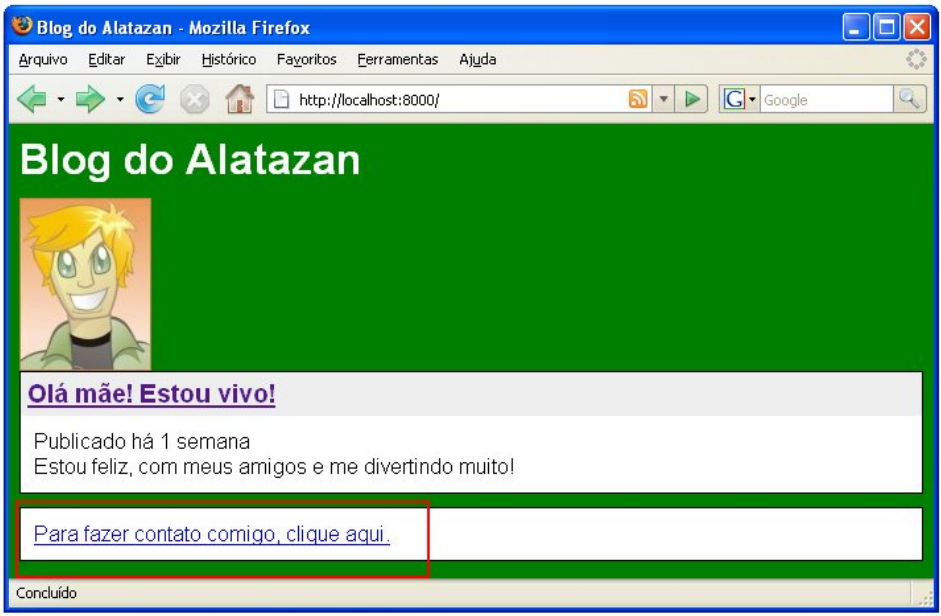
Agora o nosso link funcionou, apesar de ficar um azul perdido no verde. Vamos ajustar o estilo de nossa página para melhorar isso.

Na pasta do projeto, vá à pasta "media", abra o arquivo "layout.css" para edição e acrescente esse trecho de código ao final do arquivo:

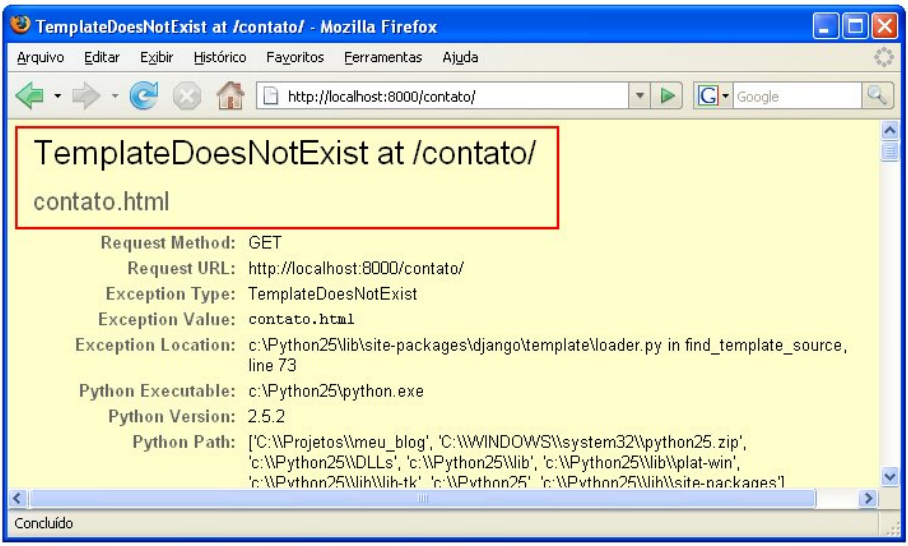
```
.rodape {
  border: 1px solid black;
  background-color: white;
  color: gray;
  margin-top: 10px;
  padding: 10px;
```

```
}
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, pressione F5 e veja como ficou:



Agora sim! E agora, ao clicar sobre o link que criamos, veja o que acontece:



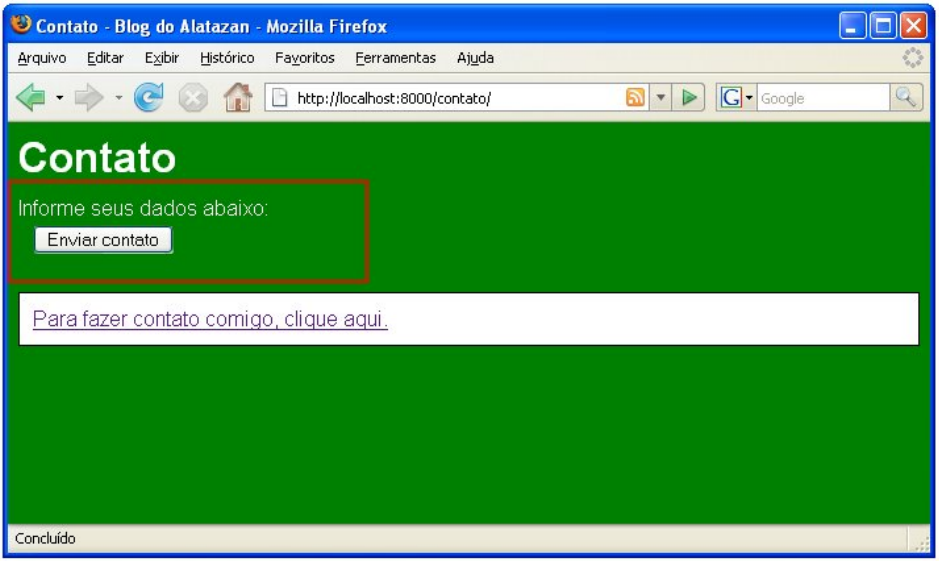
Bom, nada mais claro: a template "contato.html" não existe.

Agora, vá à pasta "templates" do projeto e crie um novo arquivo: "contato.html", com o seguinte código dentro:

```
{% extends "base.html" %}

{% block titulo %}Contato - {{ block.super }}{% endblock %}

{% block h1 %}Contato{% endblock %}

{% block conteudo %}
{% if mostrar %}<h3>{{ mostrar }}</h3>{% endif %}

Informe seus dados abaixo:

<form method="post">
  <table>
    {{ form }}

    <tr>
      <th> </th>
      <td><input type="submit" value="Enviar contato"/></td>
    </tr>
  </table>
</form>
{% endblock conteudo %}
```

Salve o arquivo. Feche o arquivo.

Ufa! Bastante coisa, não é verdade? O que fizemos aí foi declarar, além dos blocks habituais, um formulário para o envio da mensagem de contato. De volta ao navegador, veja como ficou a página de contato:



Mas como notou, só há o botão "Enviar contato". De volta ao código, veja que escrevemos uma variável {{ form }}. É esta variável que vai trazer a mágica para dentro do template. E como ela não foi declarada na nossa view, é bastante natural que aqui não apareça nada além do que vemos.

Trabalhando com os formulários dinâmicos do Django

Agora, abra o arquivo "views.py" da pasta do projeto novamente, e localize a seguinte linha:

```
def contato(request):
```

Abaixo dela, acrescente a seguinte:

```
    form = FormContato()
```

Mas isso não basta, afinal, quem é o FormContato ali? Então agora, localize esta linha:

```
from django.template import RequestContext
```

E acrescente este trecho de código:

```
from django import forms


class FormContato(forms.Form):
    nome = forms.CharField(max_length=50)
```

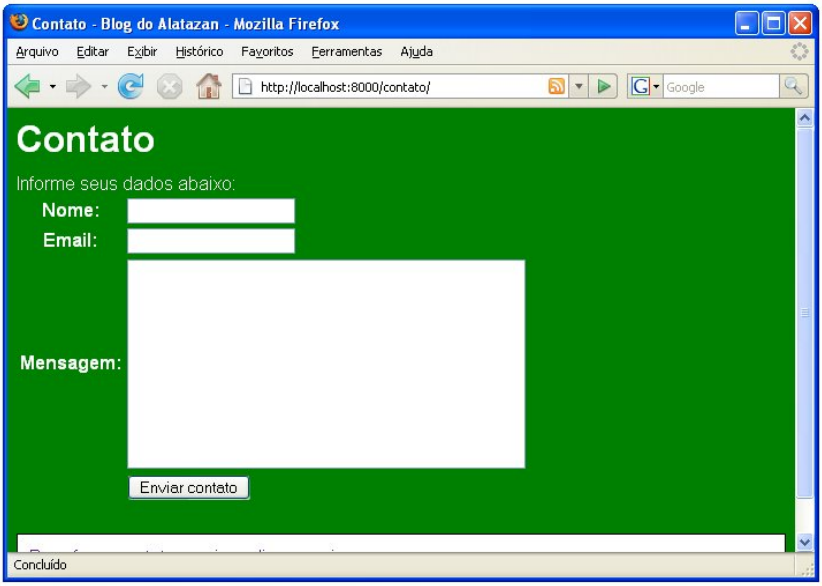
```
    email = forms.EmailField(required=False)
    mensagem = forms.Field(widget=forms.Textarea)
```

Agora o arquivo ficou assim:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from django import forms


class FormContato(forms.Form):
    nome = forms.CharField(max_length=50)
    email = forms.EmailField(required=False)
    mensagem = forms.Field(widget=forms.Textarea)


def contato(request):
    form = FormContato()
    return render_to_response(
        'contato.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

O que fizemos foi criar um formulário. Este formulário tem três campos: nome, email e mensagem. O campo "nome" é do tipo para receber caracteres, numa largura máxima de 50 caracteres. O campo "email" é do tipo para receber e-mail, mas não é requerido. Por fim, o campo "mensagem" é de um tipo livre, que será exibido como um Textarea, ou seja, uma caixa de texto maior que a habitual, que aceita muitas linhas livremente.

Salve o arquivo. Volte ao navegador e pressione F5 para atualizar a página de contato. Veja como ficou:



Veja o quanto esse recurso de formulários dinâmicos do Django é fantástico! Temos o nosso formulário sendo exibido em um piscar de olhos! Mas isso não basta, pois ele ainda não está funcionando de fato.

Pois então voltemos ao arquivo "views.py" para localizar a seguinte linha:

```
    form = FormContato()
```

Substitua essa linha pelo seguinte trecho de código:

```
    if request.method == 'POST':
        form = FormContato(request.POST)

        if form.is_valid():
            form.enviar()
            mostrar = 'Contato enviado!'
    else:
        form = FormContato()
```

Esse nosso código faz o seguinte:

• Se o método dessa request é do tipo POST, entende-se que o usuário clicou sobre o botão "Enviar contato". Isso faz sentido porque o carregamento padrão de páginas utiliza o método do tipo GET. Ou seja, somente quando o usuário clica no botão, e desde que a tag <form> do template possui um argumento

method="post", essa linha será válida;

- Neste caso, a variável "form" recebe a instância do formulário "FormContato", contendo os valores enviados no request.POST. Esses valores são aqueles que o usuário informou no formulário.

- Feito isso, o formulário passa por uma validação (form.is_valid()). É nesse momento que ele verifica se os campos requeridos foram todos preenchidos, e se todos os campos foram preenchidos corretamente de acordo com seus tipos (por exemplo: um endereço de e-mail não pode ter um formato diferente do habitual que conhecemos);

- Se a validação seguir com sucesso, é chamado o método form.enviar(), que enviará a mensagem;

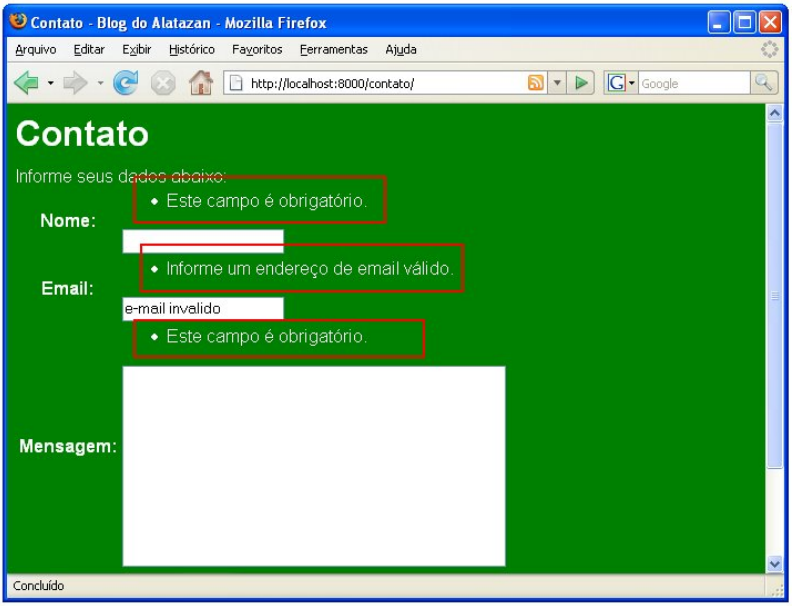
- Por fim, é mostrada uma mensagem que avisa ao usuário que a mensagem foi enviada com sucesso.

No caso de qualquer coisa dessas acima não sair da forma esperada, as devidas mensagens de erro ou validação serão exibidas.

Salve o arquivo. Volte ao navegador. E faça o seguinte:

- Preencha apenas o campo "E-mail", com um e-mail inválido
- Clique sobre o botão "Enviar contato" sem preencher os demais campos.

Veja como ficou:



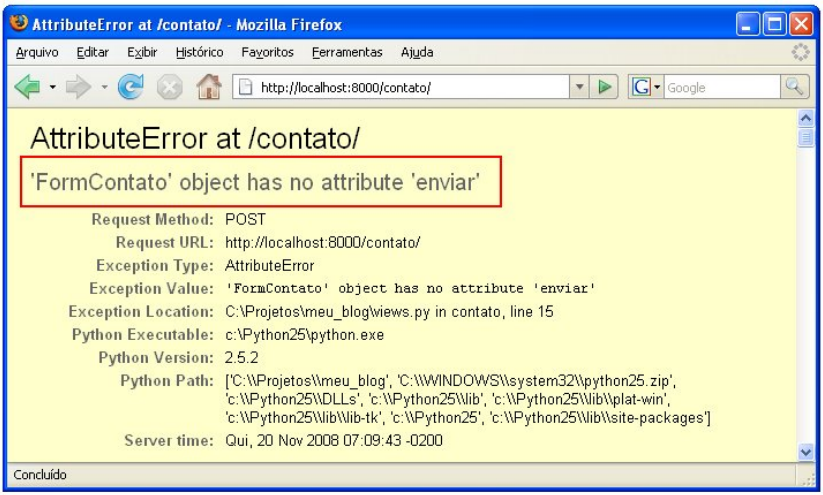
Você pode notar que os campos requeridos foram exigidos, e o formato do campo de E-mail não foi aceito.

A estética fica por conta do seu CSS.

Agora preencha todos os campos corretamente:

- Nome: Mamãe
- E-mail: mamae.do.alatazan@katara.gov
- Mensagem: Olá filhinho, sou eu usando o e-mail do trabalho indevidamente

E clique sobre "Enviar contato". Veja o que acontece:



Puxa, mas o que é isso agora?

Anime-se! Isso é um bom sinal, pois significa que chegamos ao ponto no código onde tudo foi validado e o método form.enviar() não foi encontrado.

Enviando o e-mail do contato

De volta ao arquivo "views.py", localize a seguinte linha:

```
    mensagem = forms.Field(widget=forms.Textarea)
```

Agora acrescente abaixo dela:

```
    def enviar(self):
        titulo = 'Mensagem enviada pelo site'
        destino = 'alatazan@gmail.com'
```

```
        texto = """
        Nome: %(nome)s
        E-mail: %(email)s
        Mensagem:
        %(mensagem)s
        """ % self.cleaned_data

        send_mail(
            subject=titulo,
            message=texto,
            from_email=destino,
            recipient_list=[destino],
            )
```

Agora substitua o trecho 'alatazan@gmail.com' pelo seu e-mail.

Mas afinal, o que esse método gigantesco faz?

- Ele define um título para o e-mail, que será enviado para você;
- Ele define que o destino desse e-mail será o seu e-mail;
- Ele define o texto desse e-mail com as informações preenchidas pelo usuário, usando a sintaxe de formatação de strings do Python;
- E por fim, ele envia um e-mail (função send_mail()) com esses dados.

No Python, para informar uma string com mais de uma linha, você pode usar três aspas duplas ou simples, como por exemplo """esta aqui""".

Mas o método send_mail não foi devidamente importado. Portanto, vá ao início do arquivo e acrescente a seguinte linha:

```
from django.core.mail import send_mail
```

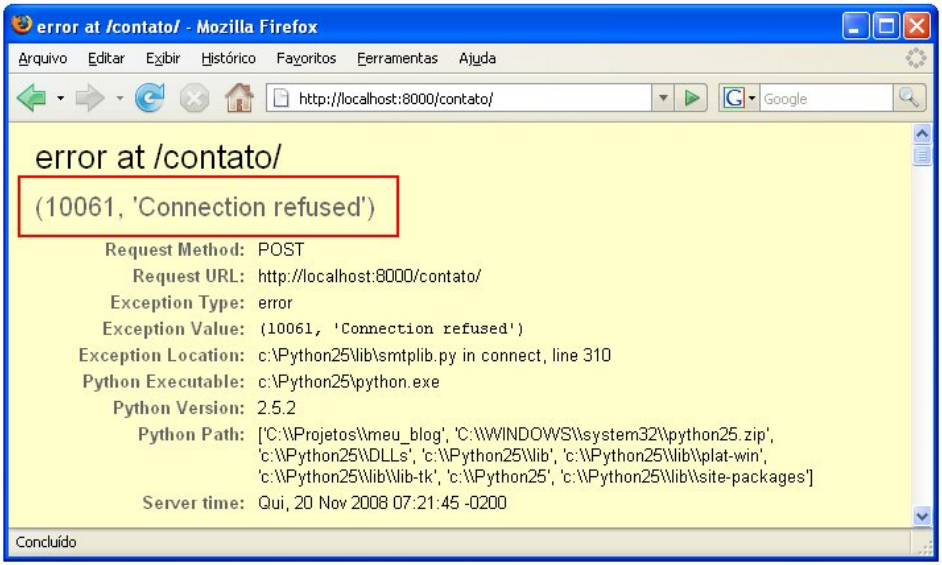
Agora, o arquivo "views.py" completo ficou assim:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from django import forms
from django.core.mail import send_mail


class FormContato(forms.Form):
    nome = forms.CharField(max_length=50)
```

```
    email = forms.EmailField(required=False)
    mensagem = forms.Field(widget=forms.Textarea)

    def enviar(self):
        titulo = 'Mensagem enviada pelo site'
        destino = 'alatazan@gmail.com'
        texto = """
        Nome: %(nome)s
        E-mail: %(email)s
        Mensagem:
        %(mensagem)s
        """ % self.cleaned_data

        send_mail(
            subject=titulo,
            message=texto,
            from_email=destino,
            recipient_list=[destino],
            )

def contato(request):
    if request.method == 'POST':
        form = FormContato(request.POST)

        if form.is_valid():
            form.enviar()
            mostrar = 'Contato enviado!'
            form = FormContato()
    else:
        form = FormContato()

    return render_to_response(
        'contato.html',
        locals(),
```

```
        context_instance=RequestContext(request),
        )
```

Salve o arquivo. Feche o arquivo. Volte ao navegador e pressione F5. Ele vai fazer uma pergunta sobre processar dados, devido ao método dessa requisição ter sido do tipo POST. Confirme a pergunta e veja o resultado:



Bom, avançamos, mas o que essa mensagem quer dizer?

```
(10061, 'Connection refused')
```

Isso acontece por que é preciso configurar seu projeto com as informações de qual servidor SMTP ele deve usar para enviar esta mensagem de e-mail.

Configurações de envio de e-mail

Para isso, abra o arquivo "settings.py" da pasta do projeto para edição, vá ao final do arquivo e acrescente o seguinte trecho de código dentro:

```
EMAIL_HOST = 'seu endereço de SMTP'
EMAIL_HOST_USER = 'seu endereço de e-mail'
EMAIL_HOST_PASSWORD = 'sua senha'
EMAIL_SUBJECT_PREFIX = '[Blog do Alatazan]'
```

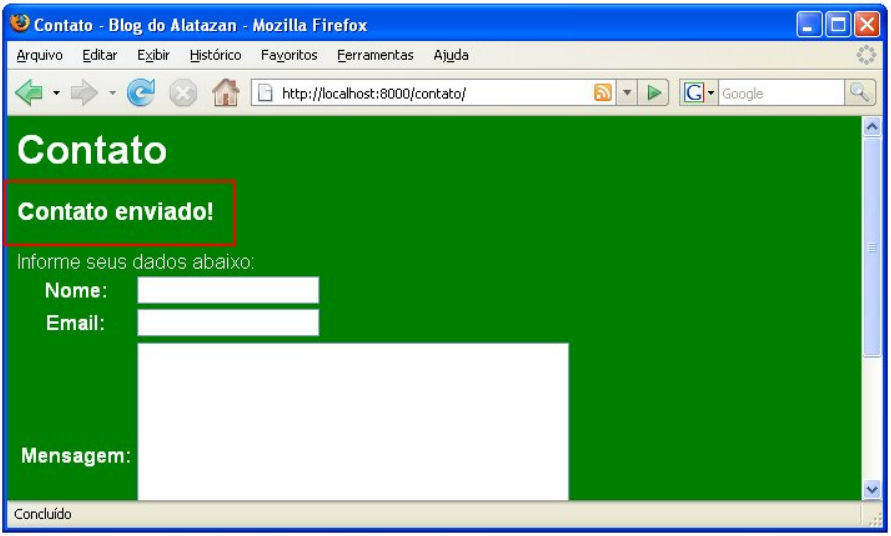
Se caso seu servidor de e-mail utilizar conexão segura, acrescente esta linha:

```
EMAIL_PORT = 587
```

Se caso seu servidor de e-mail utilizar conexão do tipo TLS, acrescente esta linha também:

```
EMAIL_USE_TLS = True
```

Agora voltando ao navegador e pressionando F5, veja o que acontece:



Pronto! Mais uma página virada em nossa caminhada!

Partindo para outra forma de receber contato...

• Caramba! Quando eu vi o formulário com os campos na página depois de escrever menos de 10 linhas de código, fiquei espantado!

• Sim, e veja, o que você fez, num resumo rápido foi isso:

  • Declarou um link para a página de contato usando a template tag {% url %};

  • Criou uma view para a página de contato;

  • Criou um novo template para a view de contato. Nele foi criado um <form>;

  • Declarou uma classe de Form e seus campos;

  • Definiu na view uma condição que verifica se o usuário clicou sobre o botão de enviar para efetuar o envio do e-mail;

- Declarou um método "enviar", que dá a ação que queremos ao form;
- Esse método formata o texto da mensagem e envia o e-mail, usando uma função do Django para isso;
- E configurou as settings para envio de e-mails.
- Ao todo foram 8 passos para cumprir a nossa missão de hoje! E de quebra ainda fez um ajuste no visual no rodapé.

Alatazan estava em êxtase, seu coração da frente clareava o peito, e isso tinha a ver com sua alegria em chegar até aqui no estudo do Django.

- Bom, já que aprendemos tanto assim hoje, amanhã vamos criar recursos para seus usuários enviarem Comentários em seu blog. Cada artigo vai ter sua própria lista de comentários!
- Show de bola!

Capítulo 11: Deixando os comentários fluírem

Alatazan estava bebendo um copo d'água no intervalo da aula, quando uma garota chegou até ele. Ela segurava um copo com refrigerante e comia um pastel.

- Oi, desculpe, mas seu braço está sujo aqui oh.

Alatazan deu uma olhada em seu cotovelo, mas a única coisa que havia ali eram as listras habituais dos katarenses. Um pouco constrangido e sem saber como explicar à moça, ele gaguejou, antes que outra garota, que parecia ser amiga dela, se aproximou rindo.

- Não acredito... não sua tonta, é mancha, de nascença, você está fazendo o menino ficar com vergonha aqui... presta atenção!

E riu um pouco mais...

Alatazan deu uma risadinha meio de meia boca e deu um jeito de sair dali... não queria entrar naquele "debate".

Depois de pensar um pouco, ele percebeu que isso tinha seu lado bom. No final das contas, essa liberdade de expressão acaba sendo positiva e bem-vinda.

Habilitando comentários no blog

Às vezes você quer que as pessoas tenham um espaço para **emitir suas opiniões** sobre aquilo que escreve em seu blog. Muitas pessoas gostam disso. E o Django oferece esse recurso de maneira muito acessível.

A coisa se resume a habilitar a aplicação *contrib* de **comentários**, adicionar uma URL e depois ajustar alguns templates. Vamos lá?

Antes de mais nada, execute seu projeto clicando duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto.

Na pasta do projeto, abra o arquivo **"settings.py"** para edição e localize a setting **INSTALLED_APPS**. Adicione a ela a seguinte linha:

```
    'django.contrib.comments',
```

Ou seja, o trecho de código da setting **INSTALLED_APPS** fica assim:

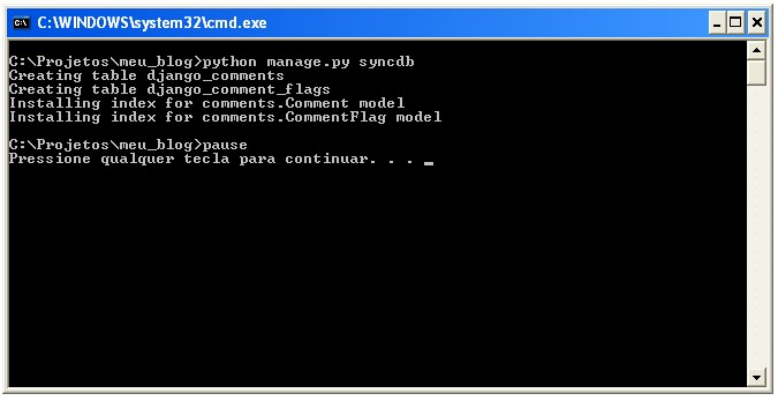
```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',
    'django.contrib.comments',

    'blog',
)
```

Temos agora uma nova aplicação no projeto: **comments**.

Salve o arquivo. Feche o arquivo.

Agora atualize o banco de dados, clicando duas vezes sobre o arquivo **"gerar_banco_de_dados.bat"** da pasta do projeto.

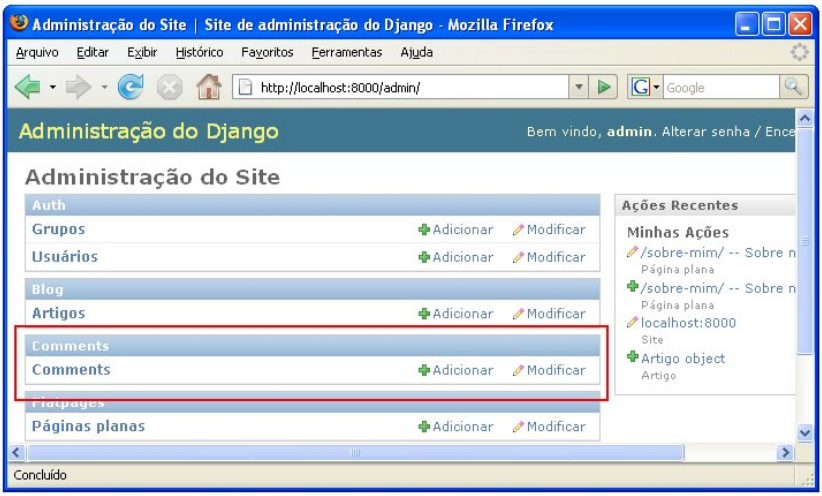
Uma janela do MS-DOS será exibida mostrando o seguinte resultado:



Ao carregar o navegador na URL do **Admin**:

```
http://localhost:8000/admin/
```

Você pode ver que há ali uma nova aplicação chamada **"Comments"**:



Devido a uma série de ajustes que ela sofreu para a **versão 1.0**, é possível que em sua versão ela ainda não esteja traduzida por completo para português brasileiro.

Mais à frente, você vai utilizar essa seção do **Admin** para administrar os comentários recebidos, pois tem hora que a paciência não é suficiente para lidar com algumas *liberdades* de certos indivíduos.

Pois agora vamos ao segundo passo: **adicionar uma URL** para a aplicação **"comments"**.

Na pasta do projeto, abra o arquivo **"urls.py"** para edição e adicione a seguinte URL ao final da função **patterns()**:

```
(r'^comments/', include('django.contrib.comments.urls')),
```

A função **include()** inclui as URLs de outro arquivo na URL em questão. Neste caso, as URLs do módulo **'django.contrib.comments.urls'** são incluídas na URL **'^comments/'**. Na prática, se no módulo **'django.contrib.comments.urls'** houver uma URL **'^post/$'**, o Django junta uma coisa à outra e entende que possui uma URL **'^comments/post/$'** apontando para tal. Isso é feito para todas as URLs encontradas no módulo incluído (**'django.contrib.comments.urls'**).

É uma solução prática para dividir as URLs quando o arquivo se torna muito extenso, e também é muito prático quando você quer separar as coisas para que suas aplicações se tornem mais flexíveis.

Agora, o arquivo **"urls.py"** ficou assim:

```
from django.conf.urls.defaults import *
```

```
from django.conf import settings


# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()


from blog.models import Artigo
from blog.feeds import UltimosArtigos


urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
         'date_field': 'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$', \
     'django.contrib.syndication.views.feed',
     {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
    (r'^media/(.*)$', 'django.views.static.serve',
     {'document_root': settings.MEDIA_ROOT}),
    (r'^contato/$', 'views.contato'),
    (r'^comments/', include('django.contrib.comments.urls')),
)
```

Salve o arquivo. Feche o arquivo.

Trabalhando as template tags no template do artigo

Agora, na pasta do projeto, abra a pasta **"blog/templates/blog"**. Dentro dela, abra o arquivo **"artigo.html"** para edição e localize a seguinte linha:

```
{% extends "base.html" %}
```

Acrescente abaixo dela a seguinte linha:

```
{% load comments %}
```

Isso vai carregar as template tags da aplicação **comments**. Agora localize esta outra linha:

```
{{ artigo.conteudo }}
```

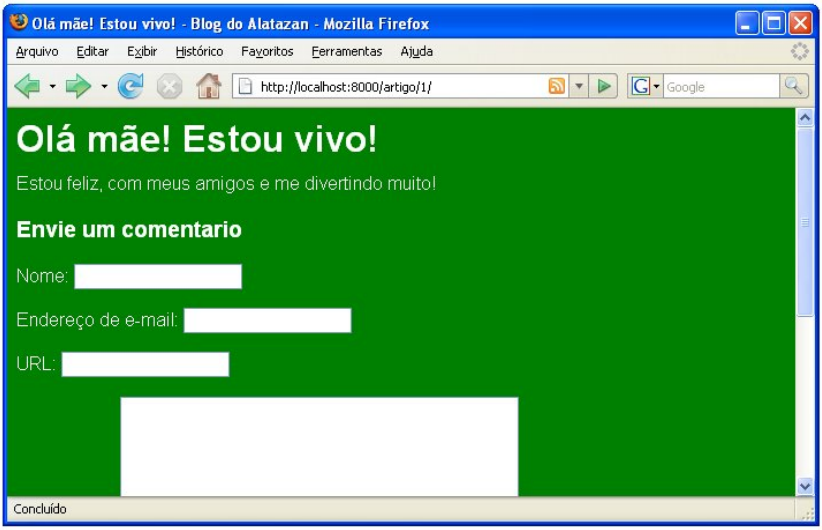
Acrescente esse trecho de código abaixo dela:

```
<div class="comentarios">
  <h3>Envie um comentario</h3>

  {% render_comment_form for artigo %}
</div>
```

Salve o arquivo. Agora vá ao navegador e localize a URL do artigo:

```
http://localhost:8000/artigo/1/
```

E veja o resultado:



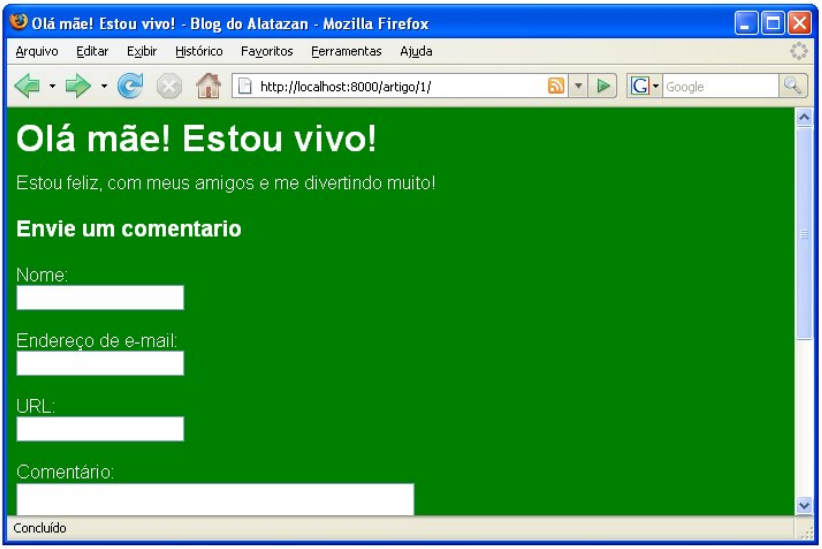
Opa, aí está um formulário de contato completo! Mas o que nós fizemos ali? A template tag **{% render_comment_form for artigo %}** gera um formulário dinâmico de comentários para o objeto **"artigo"**. Lembra-se do **Form** que trabalhamos no último capítulo?

Mas ele ficou meio deformando, não? Um pouco de **CSS** vai fazer bem pra esse layout. Da pasta do projeto, entre em **"media"**, abra o arquivo **"layout.css"** para edição e acrescente o seguinte trecho de código ao final do arquivo:

```
form label {
  display: block;
}
```

```
textarea {
  height: 100px;
}
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, atualize e veja como ficou:



Agora vamos preencher o formulário de comentário para ver o que acontece? Preencha assim:

- Nome: **Garota da Escola**
- E-mail: **garota@escola.com**
- URL: **http://escola.com/garota/**
- Comentário: **Parabéns pelo blog!**

Agora clique no botão **"Post"** e veja o resultado:

Bacana! Então o envio do comentário já funcionou! Não se preocupe, vamos mudar o visual dessa página aí já-já. Agora volte à página anterior no navegador (**ALT+Esquerda**).

Vamos agora exibir os comentários do artigo?

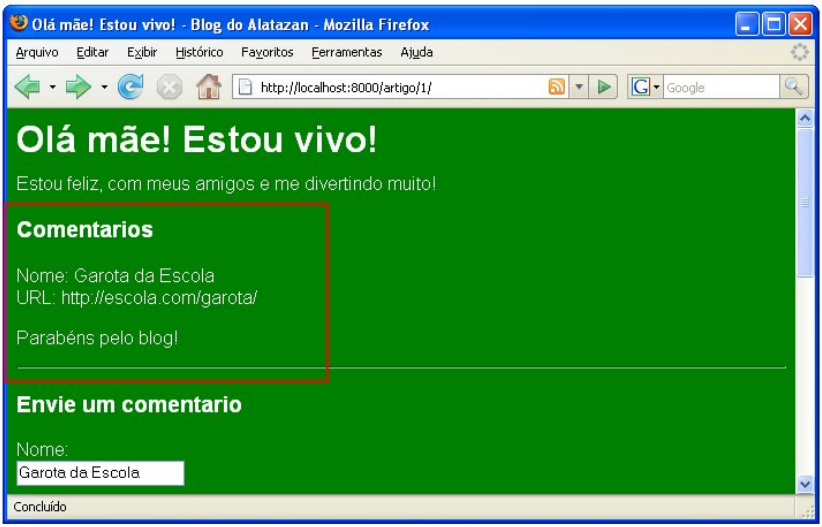
Volte ao template **"artigo.html"** que estávamos editando e localize a seguinte linha:

```
<div class="comentarios">
```

Acrescente abaixo da linha o trecho de código abaixo:

```
<h3>Comentarios</h3>


{% get_comment_list for artigo as comentarios %}
{% for comentario in comentarios %}
<div class="comentario">
  Nome: {{ comentario.name }}<br/>
  {% if comentario.url %}URL: {{ comentario.url }}
  {% endif %}<br/>
  {{ comentario.comment|linebreaks }}
  <hr/>
</div>
{% endfor %}
```

Salve o arquivo. Volte ao navegador, atualize a página do artigo e veja como ficou:



Já melhorou, não é? Pois olha o que fizemos: nós declaramos a template tag **{% get_comment_list for artigo as comentarios %}** que obtém todos os comentários que existem para o objeto **"artigo"** e os coloca na variável **"comentarios"**.

Depois, com a variável em mãos, nós fazemos um *laço* nela, usando a template tag **{% for comentario in comentarios %}**, ou seja, todo o trecho de código que está entre as template tags **{% for %}** e **{% endfor %}** será repetido a cada comentário. Se houverem 5 comentários na lista, então será repetido 5 vezes, uma para cada comentário, que é representado pela variável **"comentario"**.

Por fim, a template tag **{% if comentario.url %}** sigfinica que o trecho de código que está entre ela e a template tag **{% endif %}** só deve ser mostrado se o comentário possuir o campo **"url"** preenchido.

Agora, o template **"artigo.html"** completo está assim:

```
{% extends "base.html" %}


{% load comments %}


{% block titulo %}{{ artigo.titulo }} -
{{ block.super }}{% endblock %}


{% block h1 %}{{ artigo.titulo }}{% endblock %}
```

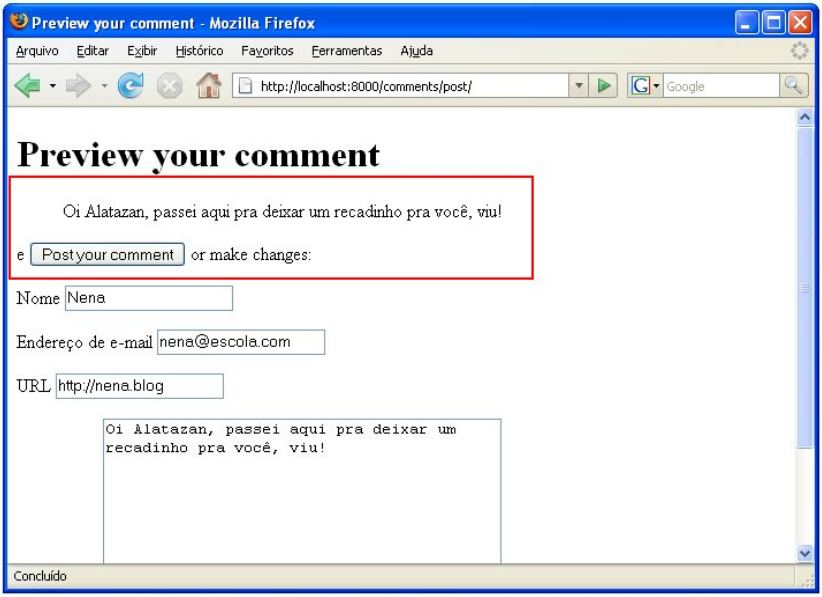
```
{% block conteudo %}
{{ artigo.conteudo }}

<div class="comentarios">
  <h3>Comentarios</h3>

  {% get_comment_list for artigo as comentarios %}
  {% for comentario in comentarios %}
  <div class="comentario">
    Nome: {{ comentario.name }}<br/>
    {% if comentario.url %}URL: {{ comentario.url }}
    {% endif %}<br/>
    {{ comentario.comment|linebreaks }}
    <hr/>
  </div>
  {% endfor %}

  <h3>Envie um comentario</h3>

  {% render_comment_form for artigo %}

</div>
{% endblock %}
```

Agora observe que no formulário do template, existe também um botão **"Preview"**. Ele faz o seguinte: ao invés de enviar o comentário, ele apenas exibe o resultado para que o usuário veja como vai ficar antes de enviar. Isso é útil para que ele tenha uma segunda chance de ajustar algo no texto.

Mas chega de explicações!

A função de pré-visualização do comentário

Escreva um novo comentário, clique sobre o botão **"Preview"** e veja o que acontece:



Primeiro, observe a caixa **destacada**. Ali será exibida a **pré-visualização** do comentário, antes de confirmar o seu envio. Esta opção não é obrigatória, você pode escolher entre trabalhar com pré-visualização ou não, e vamos ver como fazer isso em breve.

Mas o que precisamos fazer agora é dar um jeito no visual dessa página. Ela não está nada boa.

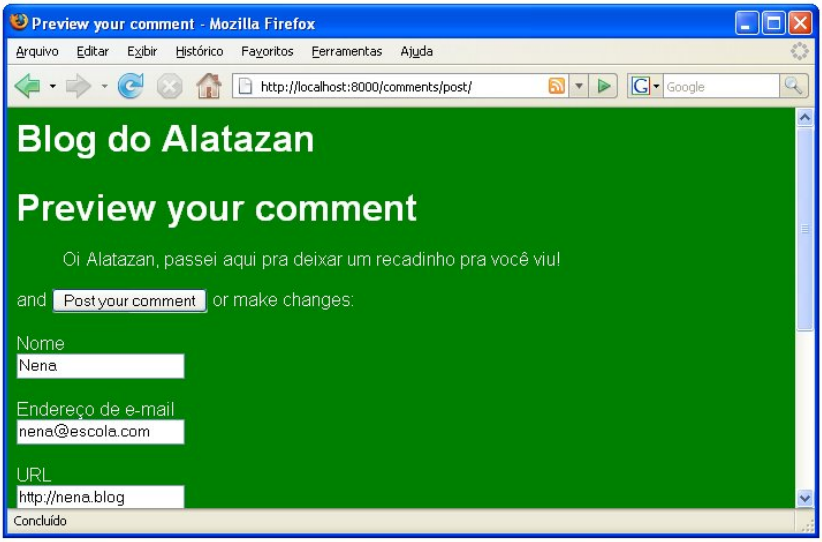
Na pasta **"templates"** da pasta do projeto, crie uma nova pasta **"comments"**. Dentro da nova pasta crie um arquivo chamado **"base.html"** e escreva dentro o seguinte código:

```
{% extends "base.html" %}


{% block titulo %}
  {% block title %}{% endblock %}
{% endblock %}


{% block conteudo %}
  {% block content %}{% endblock %}
{% endblock %}
```

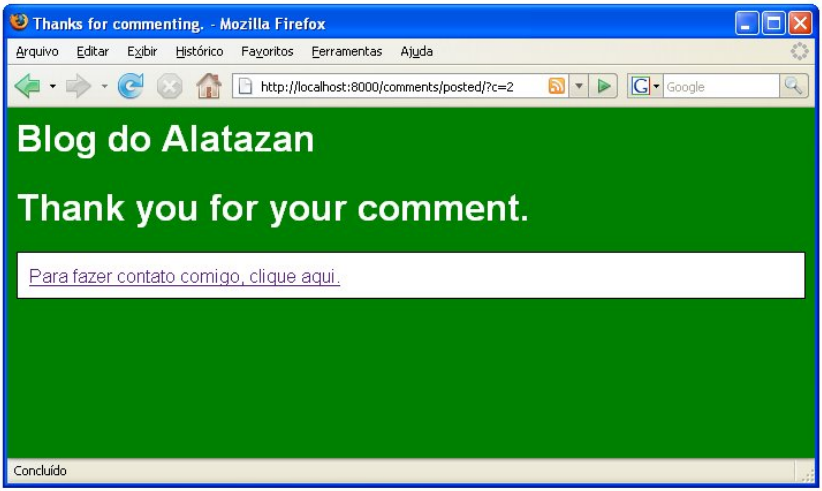
Salve o arquivo.

Mas que confusão com tantos *blocks*! Bom, o que o código acima está fazendo é levar todos os templates da contrib **"comments"** para estenderem o template do seu blog, ou seja, eles serão encaixados no seu visual.

Volte ao navegador, pressione **F5** - será feita uma pergunta sobre os dados enviados, apenas confirme - e veja como ficou:



Já melhorou, né? Agora, clique no botão **"Post"** para publicar esse comentário. Veja:



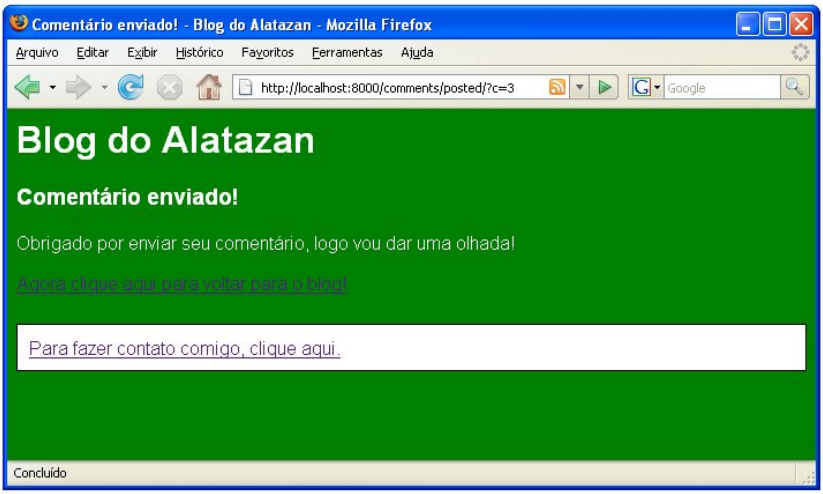
Veja que nesta página a mudança no template também surtiu efeito. Mas precisamos ajustar algo aqui. Ao menos um **link** para a página principal é necessária, certo? Não podemos deixar uma página sem meios para dar continuidade.

Depois de enviado o comentário...

Pois então vá à pasta do projeto, e abra a pasta **"templates/comments"**. Dentro dela, crie o arquivo **"posted.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}Comentário enviado! -
{{ block.super }}{% endblock %}


{% block conteudo %}
<p>
Obrigado por enviar seu comentário, logo vou dar uma olhada!
</p>


<p><a href="/">Agora clique aqui para voltar para o blog!</a></p>
{% endblock %}
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, atualize e veja como ficou:



Agora, é só lançar vários artigos e comentários para ver como a coisa funciona bem!

A caminho de colocar o site no ar...

Alatazan foi enfático, ao final:

- Bom, vamos ver o que eu fiz hoje:
  - Instalei a aplicação **'django.contrib.comments'**;
  - **Gerei** o banco de dados;
  - Adicionei a URL que inclui todas as URLs da aplicação **'comments'**;
  - Nos templates que usam template tags de comentários, tive que carregá-las com **{% load comments %}**;
  - Usei as template tags **{% render_comment_form %}** e **{% get_comment_list %}**, para gerar um formulário e obter os comentários de um artigo, respectivamente;
  - E defini um template **"base.html"** para ajustar as páginas de comentários ao **layout do site**;
- Exatamente! E você ainda fez um ajuste no CSS para dar uma quebra de linha nas tags <label>. - Cartola completou.
- Joia! Agora posso colocar esse blog no ar!
- Mesmo? Você acha que já está bom pra fazer o *deploy*? - Nena perguntou, ela também está ansiosa para ver o blog de Alatazan estrear no servidor.
- Bom, na verdade ainda quero melhorar o visual do site, mas...
- Já sei, amanhã vamos tirar o dia pra ver **HTML e CSS** o suficiente para transformar seu blog com um **bom layout**!
- Então combinado.

No próximo capítulo, vamos trabalhar novamente no layout do site, mas agora com um foco mais profissional: vamos dar um visual típico de um blog bem elaborado ao site do Alatazan, afinal, com tanto estudo, ele merece algo bem apresentável e o principal: **no ar**.

Capítulo 12: Um pouco de HTML e CSS só faz bem

Alatazan aceitou o convite de Nena para assistir à estreia de sua irmã mais nova em um desfile de moda. Estava um pouco frio, mas ainda era cedo, chegaram ao ginásio cerca de uma hora e meia antes da abertura, pois foi Nena que dirigiu para levar a irmã.

Como estavam por ali, entraram no vestiário com a menina, e acabaram vendo como a coisa acontece nos bastidores.

É engraçado como essas coisas parecem uma completa bagunça para quem não é do ramo. Panos, fitas, faixas, coisas coloridas, arames e toda sorte de agulhas e linhas eram só uma fração da variedade de coisas ali, e no começo as modelos se pareciam mais com atrizes de picadeiro.

O cheiro de cola imperava no ar, e havia também um calorzinho com cheiro de ferro de passar ligado, do lado direito, próximo à cortina.

"Neninha", como Alatazan chama carinhosamente a irmã de Nena, seria uma das últimas a participar e eles ficaram lá dentro o suficiente para ver as primeiras saírem do vestiário e caminharem para a passarela.

O interessante é que depois de toda a arrumação, em questão de segundos a coisa parece sair de uma situação incompreensível para uma apresentação louvável. A moça que antes parecia um papagaio depenado, já estava agora refinada, delicada, cheia de glamour.

Ahh, já ia esquecendo... foi nesse dia que Nena e Alatazan estudaram XHTML e CSS e o resultado foi muito interessante, acompanhe...

Tempo para o (X)HTML e o CSS

No capítulo 7, trabalhamos com templates, e tivemos um contato próximo com o HTML.

No capítulo 8, trabalhamos com arquivos estáticos, e lá usamos um pouco de CSS.

Mas agora, o que vamos tratar não é o conhecimento do sistema de templates, e

muito menos a disposição de arquivos estáticos. Estamos falando de apresentação visual, a roupagem estética que o site precisa receber antes de ser apresentado ao usuário.

Não é a nossa intenção fazer desse um trabalho visual para ganhar algum prêmio, e sim, conhecer um pouco do assunto.

Antes de mais nada, execute seu projeto, clicando duas vezes no arquivo **"executar.bat"** da pasta do projeto.

Pois bem, vamos começar pelo template mais básico de nosso site. Aquele que todos extendem: o **"base.html"**, da pasta **"templates"** do projeto. Abra esse arquivo para edição e o modifique para ficar assim:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns= "http://www.w3.org/1999/xhtml"
      xml:lang="pt-br"
      lang="pt-br"
>
 <head>
  <title>{% block titulo %}Blog do Alatazan{% endblock %}</title>

  {% block meta %}
  <meta http-equiv="Content-type"
        content="text/html;
        charset=utf-8" />
  <meta http-equiv="Content-Language" content="pt-br" />
  <meta name="keywords"
        content="Python, Django, Alatazan, Katara, blog" />
  <meta name="description" content="Este é o blog do Alatazan, um
katarense aprendendo Django." />
  {% endblock meta %}

  {% block feeds %}
  <link rel="alternate"
    type="application/rss+xml"
    title="Ultimos artigos do Alatazan"
    href="/rss/ultimos/" />
```

```
 {% endblock feeds %}

 {% block style %}
 <link rel="stylesheet"
       type="text/css"
       href="{{ MEDIA_URL }}layout.css"/>
 {% endblock style %}

 {% block scripts %}
 {% endblock scripts %}
</head>

<body>
 <div id="tudo">
  <div id="topo">
   {% block topo %}
   <div id="foto"> </div>
   Blog do Alatazan
   {% endblock topo %}
  </div>

  <div id="menu">
   {% block menu %}
   {% spaceless %}
   <ul>
    <li><a href="/">Pagina inicial</a></li>
    <li><a href="/sobre-mim/">Sobre mim</a></li>
    <li><a href="{% url views.contato %}">Contato</a></li>
   </ul>
   {% endspaceless %}
   {% endblock menu %}
  </div>

  <h1>{% block h1 %}{% endblock %}</h1>
```

```
    <div class="corpo">
     {% block conteudo %}{% endblock %}
    </div>

    {% include "rodape.html" %}

   </div>
  </body>
</html>
```

Salve o arquivo. Assegure-se de que está salvando o arquivo no padrão **UTF-8**. Mas quanta coisa hein?

Como pode notar, modificamos o template em grande parte, portanto, vamos falar das mudanças por partes.

HTML ou XHTML?

A principal mudança estrutural que fizemos ali foi que saímos do **HTML para o XHTML**. Isso aconteceu por causa dessas cinco linhas:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="pt-br"
      lang="pt-br">
```

O XHTML é uma **evolução** do HTML, que obedece a regras melhor definidas. Isso a torna potencialmente mais rápida e melhor compreendida por navegadores e mecanismos de busca.

As principais novidades que o XHTML oferece são:

- A página deve iniciar com uma declaração de **DOCTYPE**;
- A tag **<html>** deve possuir um **namespace** e definição de idioma;
- Ela é *case sensitive*, ou seja, letras maiúsculas e minúsculas são consideradas diferentes;
- Todas as tags devem ser declaradas em letras minúsculas (ex.: **<body>** ao invés de **<BODY>**);
- Todas as tags vazias devem ser fechadas com uma barra ao final (ex.: **
**

ao invés de **
**);

• Todas as tags preenchidas devem ser fechadas (ex.: **<option>teste</option>** ao invés de **<option>teste**);

• Todos os atributos das tags devem ser abertos e fechados com aspas duplas (ex.: **<p align="center">** ao invés de **<p align=center>**).

DOCTYPE

O **DOCTYPE** indica qual é o tipo da página. Cada tipo implica em um conjunto de regras diferentes que serão aplicadas à página quando esta for renderizada.

Os tipos existentes são:

• **Transitional** - é o tipo mais flexível. Permite tags HTML que são marcadas como *deprecated* (marcadas como descontinuadas) e permite também páginas que não usem CSS da maneira devida;

• **Strict** - é o tipo mais rígido. Aplica todas as regras "ao pé da letra";

• **Frameset** - aplica regras específicas para *frames* e também possui a mesma flexibilidade da **Transitional**;

Namespace e idioma

O padrão que definimos aponta o *XML namespace* "http://www.w3.org/1999/xhtml", que define o documento como compatível com o padrão de XHTML definido pelo W3C em 1999. Além disso, definimos o idioma como **"pt-br"**

Bloco "meta"

Você viu que definimos um **{% block meta %}**? Pois bem, é ali que vamos descrever tags HTML de meta-definições. As que definimos agora são:

```
<meta http-equiv="Content-type"
      content="text/html;
      charset=utf-8" />
```

Esta acima define o padrão de caracteres **utf-8**, ou seja, **Unicode**, o conjunto de caracteres que suporta os mais importantes alfabetos do mundo.

```
<meta http-equiv="Content-Language" content="pt-br" />
```

Esta outra acima define o idioma do conteúdo como **"português brasileiro"**.

```
<meta name="keywords"
      content="Python, Django, Alatazan, Katara, blog" />
```

Esta outra acima define as palavras-chave para busca. O ideal é informar entre **5 e 10 palavras-chave**, pois isso torna sua página mais consistente no momento de ser indexada por mecanismos de busca como o Google.

Não invente de colocar ali 20 ou 30 palavras pensando que isso vai fazer sua página ficar "mais importante". Uma atitude dessa faz as palavras contidas perderem seu "valor" no momento da busca, e o tiro sai pela culatra.

```
<meta name="description" content="Este é o blog do Alatazan, um
katarense aprendendo Django." />
```

Esta acima define a descrição da página. Os mecanismos de busca (como o Google) a utilizam para sua descrição quando sua página é listada nos resultados de uma busca.

Blocos "feeds", "style" e "scripts"

Cada um desses três será usado para um fim específico: Feeds, Estilos e JavaScripts, respectivamente.

Divisões do layout

div "tudo"

Com esta **div**, criamos um contêiner capaz de abranger toda a página, assim podemos delimitar a página para ficar mais amigável a resoluções de tela menores, e também com uma aparência mais agradável.

div "topo"

É o novo contêiner para o topo da página, onde estará a identificação do site, presente em todas as páginas.

div "menu"

É o novo contêiner para o menu geral do site. dentro dela há uma **lista não-numerada** (**<ul>**), que tem cada um de seus itens (**<li>**) como sendo um item do menu.

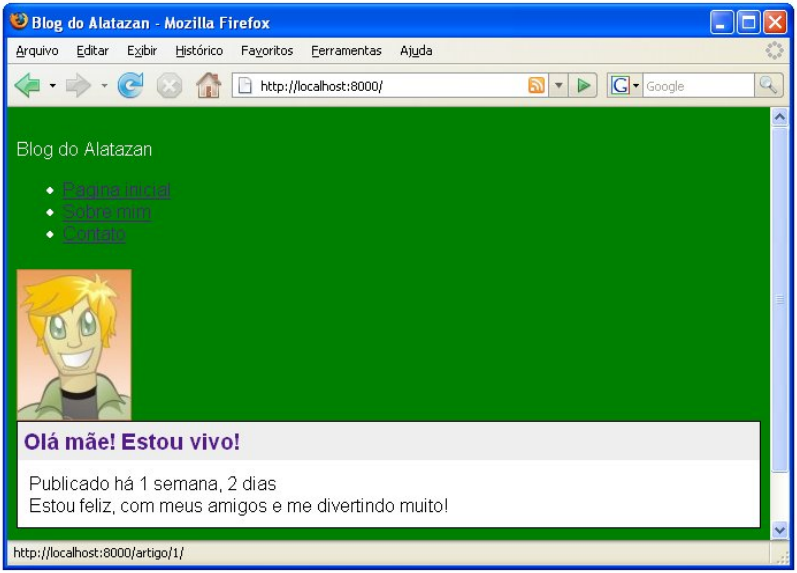
div "corpo"

É o novo contêiner para o conteúdo da página.

Bom, agora que tudo foi esclarecido, vá ao navegador no seguinte endereço:

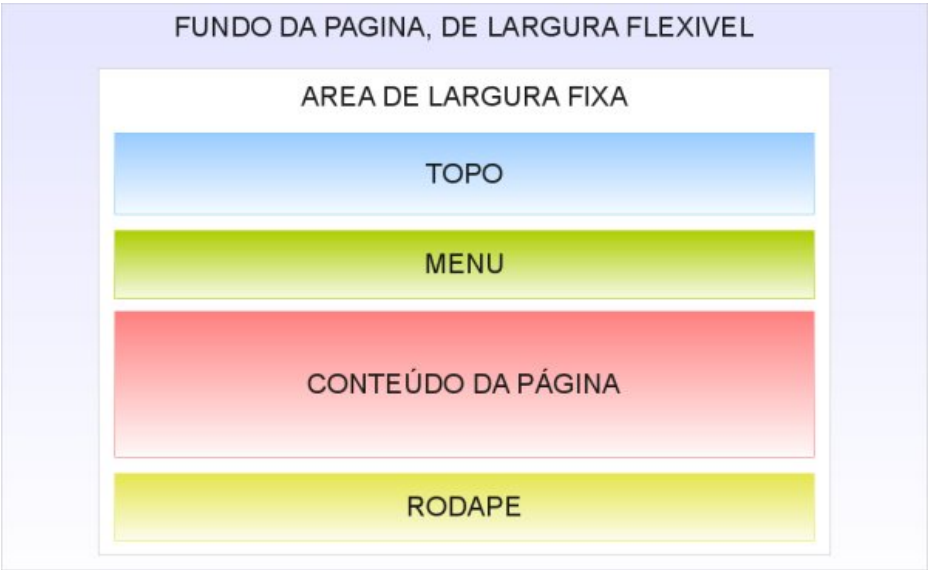
```
http://localhost:8000/
```

E veja como ficou:



Bom, está na cara que não mudou muita coisa né? Mas agora vamos à mágica, e é aí que você precisa observar com maior atenção.

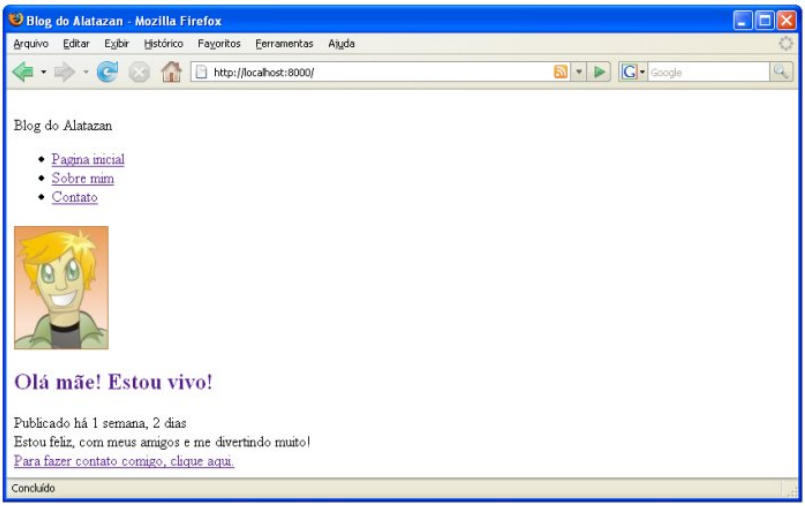
Da forma que estruturamos o **HTML**, podemos ajustar nosso layout para ficar assim:



Para fazê-lo, vá à pasta do projeto e abra a pasta **"media"**. Dali, abra o arquivo **"layout.css"** para edição.

Trabalhando no estilo da página

Vamos adotar uma atitude radical: limpe o arquivo, **delete todo seu conteúdo**. Salve o arquivo, volte ao navegador e veja como ficou:

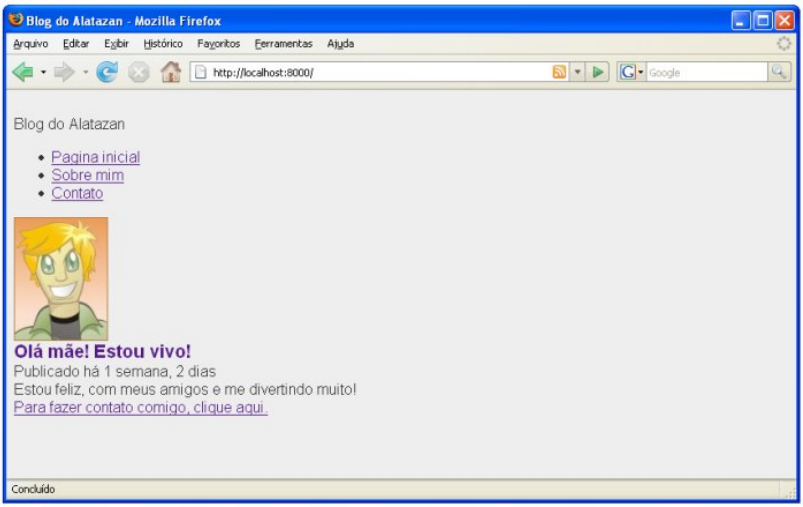


Completamente sem estilo CSS, certo? Ok, agora acrescente o seguinte trecho de código ao arquivo de CSS:

```
body {
  font-family: arial;
  background-color: #eee;
  color: #333;
}

h1 {
  margin: 10px 20px 10px 20px;
}

h2 {
  margin: 0;
  font-size: 1.2em;
```

```
}
```

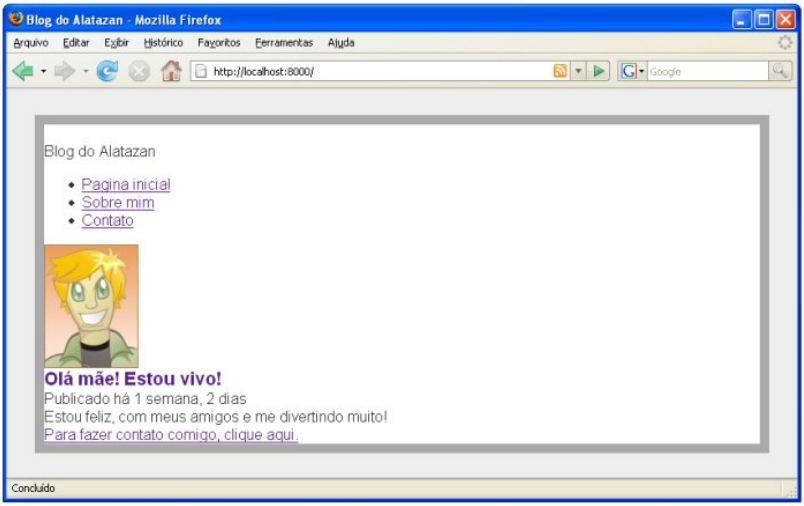
Salve o arquivo, atualize o navegador e veja como ficou:



Mudança pequena? Agora acrescente este trecho de código:

```
/* Layout */


#tudo {
  position: absolute;
  background-color: white;
  margin: 20px 0 20px -390px;
  width: 760px;
  left: 50%;
  border: 10px solid #aaa;
}
```

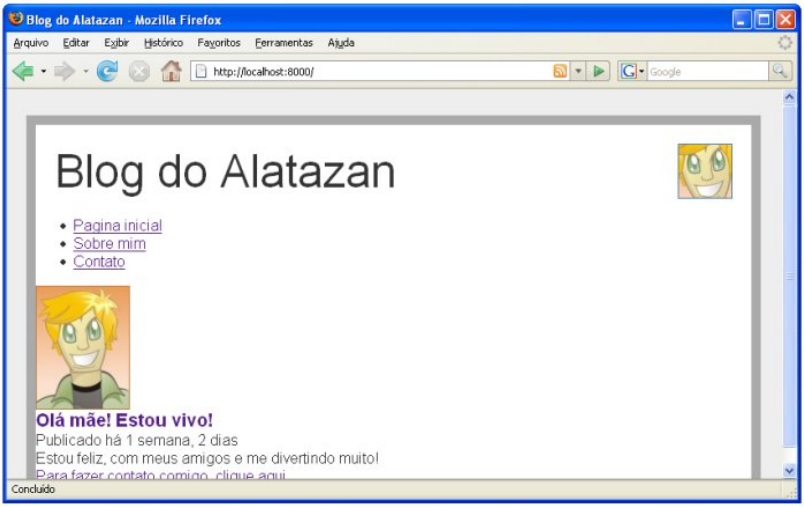
Salve o arquivo, atualize o navegador e veja como ficou:



Agora já temos uma caixa fixa no centro, certo? Vamos agora acrescentar este trecho de código:

```
#topo {
  font-size: 3em;
  margin: 20px;
}

#topo #foto {
  float: right;
  background-image: url(/media/foto.jpg);
  background-repeat: no-repeat;
  background-position: -20px -30px;
  width: 56px;
  height: 56px;
  border: 1px solid #47a;
}
```

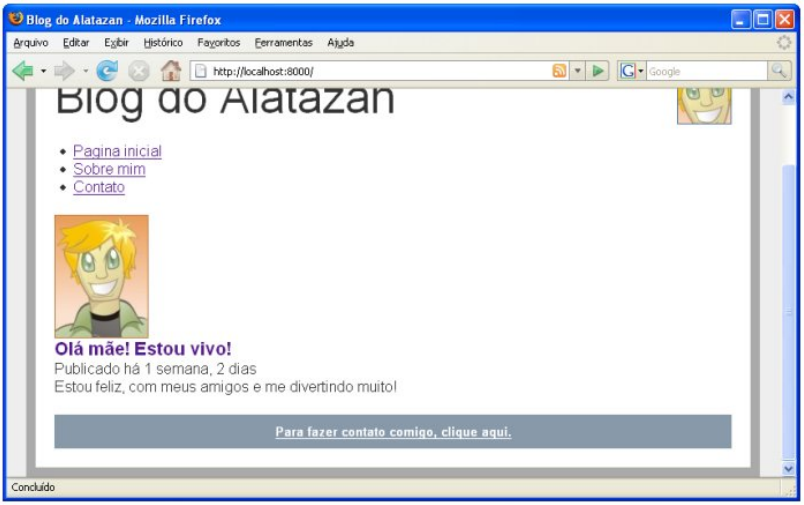
Salve o arquivo, atualize o navegador e veja como ficou:



Agora já temos o topo da página! Agora acrescente mais este:

```
.corpo {
  margin: 20px 20px 0 20px;
}

.rodape {
  clear: both;
  overflow: hidden;
  background-color: #89a;
  margin: 20px;
  font-size: 0.8em;
  color: white;
  padding: 10px;
  text-align: center;
}

.rodape a {
  color: white;
}
```

Salve o arquivo, atualize o navegador e veja como ficou:

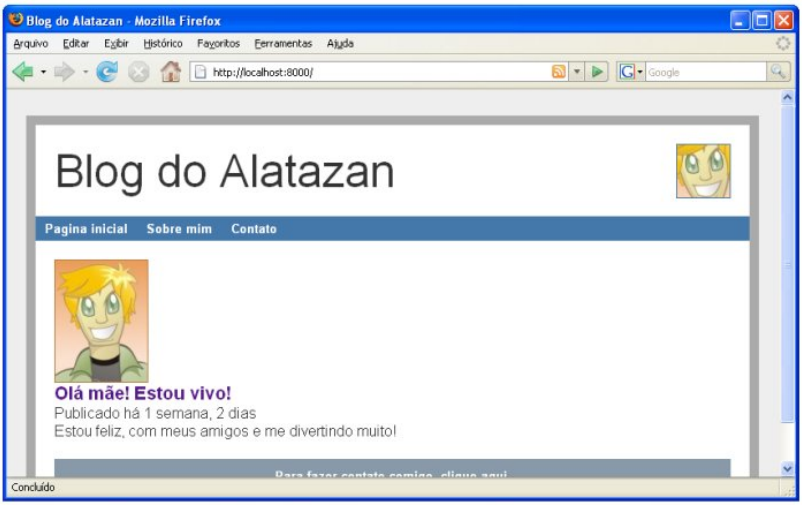


Pronto, nosso **conteúdo** e **rodapé** também estão resolvidos! Agora acrescente mais este trecho de código ao CSS:

```
/* Menu principal */


#menu {
  clear: both;
  overflow: hidden;
  background-color: #47a;
  margin-top: 10px;
  height: 26px;
  font-size: 0.8em;
  font-weight: bold;
  color: white;
}


#menu ul {
  margin: 0;
  padding: 0;
}


#menu ul li {
```

```
  float: left;
  list-style: none;
  padding: 5px;
}

#menu ul li a:link, #menu ul li a:visited, #menu ul li a:active {
  padding: 5px;
  text-decoration: none;
  color: white;
}

#menu ul li a:hover {
  background-color: #79b;
  color: white;
}
```

Salve o arquivo, atualize o navegador e veja como ficou:

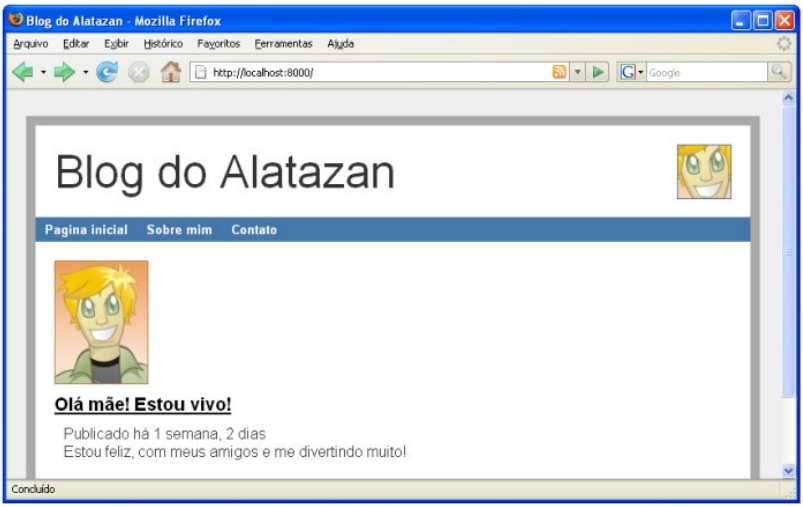


Puxa, o nosso menu já virou outro! Agora acrescente mais esse trecho de código:

```
/* Artigo do blog */
```

```
.artigo h2 {
  text-decoration: underline;
}

.artigo {
  margin: 10px 0 10px 0;
}

.artigo a {
  color: black;
}

.conteudo {
  padding: 10px;
}
```

Salve o arquivo, atualize o navegador e veja como ficou:



Notou a mudança no artigo?

Por fim, acrescente mais esse trecho:

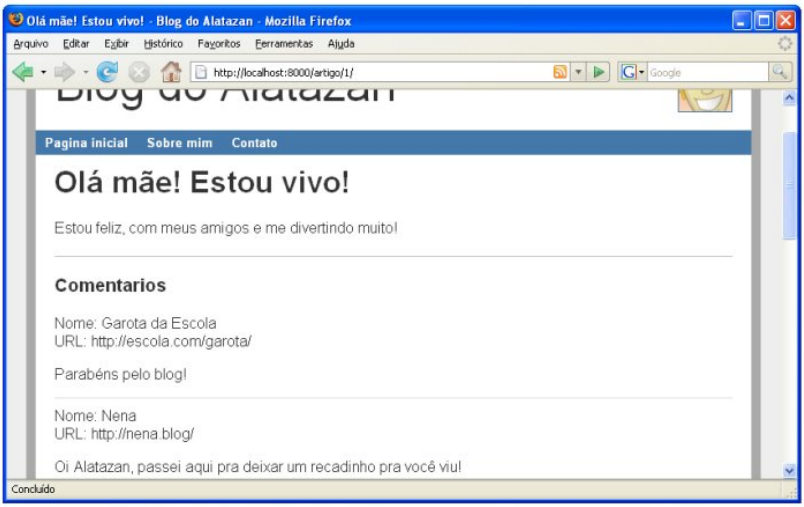
```
/* Formularios */
```

```
form label {
  display: block;
}

textarea {
  height: 100px;
}

th {
  text-align: right;
  padding-right: 5px;
}

/* Comentarios */

.comentarios {
  border-top: 1px solid silver;
  margin-top: 20px;
}

.comentarios hr {
  border-width: 0;
  height: 1px;
  border-top: 1px solid #ddd;
}
```

Salve o arquivo. Feche o arquivo. Carregue no navegador o endereço do artigo:

```
http://localhost:8000/artigo/1/
```

E veja como ficou:



Pois bem, agora veja como está o nosso CSS depois de tudo:

```
body {
  font-family: arial;
  background-color: #eee;
  color: #333;
}

h1 {
  margin: 10px 20px 10px 20px;
}

h2 {
  margin: 0;
  font-size: 1.2em;
}

/* Layout */

#tudo {
  position: absolute;
```

```
  background-color: white;
  margin: 20px 0 20px -390px;
  width: 760px;
  left: 50%;
  border: 10px solid #aaa;
}

#topo {
  font-size: 3em;
  margin: 20px;
}

#topo #foto {
  float: right;
  background-image: url(/media/foto.jpg);
  background-repeat: no-repeat;
  background-position: -20px -30px;
  width: 56px;
  height: 56px;
  border: 1px solid #47a;
}

.corpo {  margin: 20px 20px 0 20px;  }

.rodape {
  clear: both;
  overflow: hidden;
  background-color: #89a;
  margin: 20px;
  font-size: 0.8em;
  color: white;
  padding: 10px;
  text-align: center;
}
```

```
.rodape a {
  color: white;
}

/* Menu principal */

#menu {
  clear: both;
  overflow: hidden;
  background-color: #47a;
  margin-top: 10px;
  height: 26px;
  font-size: 0.8em;
  font-weight: bold;
  color: white;
}

#menu ul {
  margin: 0;
  padding: 0;
}

#menu ul li {
  float: left;
  list-style: none;
  padding: 5px;
}

#menu ul li a:link, #menu ul li a:visited, #menu ul li a:active {
  padding: 5px;
  text-decoration: none;
  color: white;
}
```

```
#menu ul li a:hover {
  background-color: #79b;
  color: white;
}

/* Artigo do blog */

.artigo h2 {
  text-decoration: underline;
}

.artigo {
  margin: 10px 0 10px 0;
}

.artigo a {
  color: black;
}

.conteudo {
  padding: 10px;
}

/* Formularios */

form label {
  display: block;
}

textarea {
  height: 100px;
}
```

```
th {
  text-align: right;
  padding-right: 5px;
}

/* Comentarios */

.comentarios {
  border-top: 1px solid silver;
  margin-top: 20px;
}

.comentarios hr {
  border-width: 0;
  height: 1px;
  border-top: 1px solid #ddd;
}
```

Os macetes mais importantes do CSS

Resumindo a história, os macetes mais importantes do CSS para quem está começando são:

- **Evite usar tabelas**, com todas as suas forças, para o que não é *de fato* uma matriz de dados tabulares. Abra mão disso somente quando não houver mais alternativas;
- Evite *padding* quando há um elemento dentro de outro. É melhor que o *padding* do elemento pai seja **zero** e a margem do elemento filho faça o trabalho do espaçamento;
- Use **overflow: hidden** para garantir que aquele elemento não ficará deformado quando algo dentro dele fizer aquela força danada para sair;
- Use **clear: both** para garantir que aquele elemento vai ficar sozinho na linha onde ele está posicionado.
- Use **float: left** ou **float: right** para posicionar um elemento à esquerda ou à direita respectivamente, ainda que ele não seja *inline*. E faça isso com ele antes dos demais elementos no código HTML. Esse é o segredo do menu horizontal em lista;

- Use unidade **"em"** para tamanhos de fonte;

- Caso queira exibir somente uma parte retangular de uma imagem, crie uma **div** com a imagem em *background*, depois posicione a imagem onde quiser.

Ajustando o rodapé

Bom, agora que temos um CSS bem elaborado, vamos fazer um último ajuste. Na pasta **"templates"** da pasta do projeto, abra o arquivo **"rodape.html"** para edição e o modifique para ficar assim:

```
<div class="rodape">
Obrigado por passar por aqui. Volte sempre.
</div>
```

Salve o arquivo. Feche o arquivo. Agora nosso rodapé tem uma cara um pouco mais de... **rodapé**!

Satisfeito com o visual, mas de olho no Ballzer

Chegando em casa, Alatazan abria a porta quando uma bola metálica surgiu à sua frente, do nada!

A bola metálica tinha uma fileira de dentes, braços compridos, olhos esbugalhados e aquele sorriso habitual.

- Tcharam! Poc poc! Ziiig poc poc!

Alatazan por fim confirmou o que vinha suspeitando há alguns dias. **Ballzer** estava ali, mais metálico do que nunca, e provavelmente aprontando.

Depois de dar um abraço destrambelhado em Alatazan, Ballzer foi ao fundo do quarto e trouxe um robozinho pequeno, desses feitos por pesquisadores do Planeta Terra. O robozinho não fazia muito mais do que girar para lá e para cá, e parecia que Ballzer havia adotado-o como mascote, boneca ou algo do gênero.

- Ballzer, onde você conseguiu isso?

No outro dia de manhã, desiludido e sem seu sorriso habitual, Ballzer levou Alatazan à casa de Cartola, de onde havia tirado o robô.

Depois da explicação constrangedora e sem-graça de Alatazan, Cartola parecia estar doido pra ele terminar com aquilo logo pra lhe mostrar algo mais interesse.

- Tá bom cara, mas olha, sabe em quê esse robô foi programado? Python, cara! **Python**! Eu consegui isso lá no laboratório da faculdade, com um cara meio maluco, e trouxe pra dar uma olhada. Vamos tirar um tempo hoje pra fuçar nisso e ver o que a gente aprende aqui!

Capítulo 13: Um pouco de Python agora

Cartola estava excitado com o fato do robô ter sido programado em Python, mas estava achando estranhas algumas linhas de código que tinham erros de lógica incomuns.

Alatazan já desconfiou logo:

- **Ballzer**, você andou futricando na programação desse robô? Vai me dizer que já está me arrumando encrenca?

Ballzer baixou seus olhos, tratou de se esconder um pouco mais atrás da cama de Cartola e assentiu arrependido.

- Você está me dizendo que esse chimpanzé metálico em forma de polvo, programa em **Python**? - Cartola perguntou um pouco espantado.

- Programar não programa, sabe... mas ele adora mexer nas mesmas coisas que eu estou mexendo...

- Coitadinho, não fala assim com ele, ele é seu maior fã, Alatazan.

Nena disse isso mas Alatazan reagiu com aquela cara de "não piore as coisas, por favor".

- Bom, então vamos aproveitar esse momento pra falar sobre algumas **coisas fundamentais do Python**. É importante saber essas coisas antes de continuar com o estudo do Django, sabia?

Python, uma linguagem maior que o Django...

Um dia desses Cartola comentou com Alatazan que **"o Python é a linguagem que manda os paradigmas para o espaço"**. É bem provável que Ballzer ouviu isso, pois *"mandar as coisas para o espaço"* é uma expressão particularmente especial para ele. O que acontece é que Ballzer não compreende muito bem essa coisa de metáforas, figuras, interpretações e tal, e deve ter entendido que aquela era a ferramenta ideal para um *oba-oba*.

Não, não... não é isso. O Python não é exatamente uma linguagem recomendada

para quem gosta de gambiarras, muito pelo contrário, ela é indicada a pessoas que gostam de se divertir programando, mas fazendo as coisas do jeito certo.

Mas de fato, o Python rompe paradigmas, e que pode ser chamada também de **"linguagem multi-paradigmas"**.

Orientada a objetos ou funcional?

Python é as duas, mas também permite orientação a aspectos, programação procedural e outras práticas de outros paradigmas.

Para desenvolver em Django usa-se muito funções isoladas (paradigma procedural), definição e instanciamento de classes (orientação a objetos), *decorators* (orientação a aspectos e funcional), etc.

Mas ainda assim, o principal paradigma que o Django trabalha no Python, é de fato a programação **orientada a objetos**.

Interpretada ou Compilada?

O Python também trabalha com ambos os paradigmas. Quer proteger o código? Leve somente os arquivos **.pyc**. Isso vai dificultar o suficiente para quem quiser explorar suas regras de negócio. Quer trabalhar diretamente nos arquivos de código-fonte? Basta levar os arquivos **.py**.

Um código escrito em Python pode ser interpretado de várias formas:

- a forma tradicional, tecnicamente chamada **CPython**, é a forma como temos trabalhado neste estudo;
- o shell interativo do Python, é uma interpretação dinâmica em *shell* semelhante ao *prompt* do MS-DOS, onde é possível declarar objetos, classes, funções e fazer outras coisas de forma *on-line*;
- a compilação para classes **Java** (arquivos **.class**), usando **Jython**, para ser executado em máquina virtual **JVM**;
- a compilação para *bytecode* **.Net**, usando **IronPython**, para ser executado em máquina virtual .Net;
- e outras formas de compilação e/ou interpretação para **dispositivos de eletrônica e celulares**.

Windows, Linux ou Mac?

Python também é multi-paradigma nesse sentido. Ele roda em Windows, Linux, MacOS, Unix, Solaris e outros sistemas operacionais menos conhecidos.

Para saber de outras plataformas onde o Python roda, veja esta página da Web:

```
http://www.python.org/download/other/
```

Aqui, é preciso manter alguns cuidados para evitar problemas ao, por exemplo, executar em um servidor Linux um programa criado e testado em Windows.

As diferenças não são grandes, mas é importante ficar de olho nelas. Um caso clássico é o do separador de caminhos de arquivos. No Windows o separador é uma **barra invertida** (\ ), nos demais é uma **barra comum** (/). Neste caso, você pode usar sempre a barra comum ou o elemento **os.sep**.

Mas raramente essas diferenças chegam a dar dores de cabeça, pois o Python já faz a maior parte para você.

Codificação de caracteres do arquivo

Em uma pasta qualquer, crie um novo arquivo com o nome **"testar_caracteres.py"** com o seguinte código dentro:

```
def imprimir_duas_vezes(texto):
    # Imprime o texto uma vez
    print texto


    # É aqui que imprime outra vez
    print texto


imprimir_duas_vezes('Tarsila do Amaral')
```

Salve o arquivo. Vá **Prompt do MS-DOS** (ou no **Console** do Linux), localize a pasta onde você salvou o arquivo e execute:

```
python testar_caracteres.py
```

Será exibida a seguinte mensagem de erro:

```
File "testar_caracteres.py", line 5

SyntaxError: Non-ASCII character '\xc9' in file
testar_caracteres.py on line 5,

but no encoding declared; see http://www.python.org/peps/pep-
0263.html for details
```

Isso acontece porque o seu comentário da linha 5 possui um **"É"** acentuado, veja:

```
    # É aqui que imprime outra vez
```

Mesmo que salve o arquivo no formato **UTF-8** (o que já é recomendável), isso

não irá se resolver. É preciso adicionar o seguinte código ao arquivo, na primeira linha:

```
# -*- coding: utf-8 -*-
```

Ou seja, agora o arquivo ficou assim:

```
# -*- coding: utf-8 -*-


def imprimir_duas_vezes(texto):
    # Imprime o texto uma vez
    print texto


    # É aqui que imprime outra vez
    print texto


imprimir_duas_vezes('Tarsila do Amaral')
```

Salve o arquivo. Execute novamente:

```
python testar_caracteres.py
```

Agora veja o resultado:

```
Tarsila do Amaral
Tarsila do Amaral
```

Isso é importante, pois o Python foi criado em solo **americano** por um **holandês** e é utilizado por pessoas do mundo inteiro, o que inclui **brasileiros**, **gregos**, **russos**, **chineses**, **japoneses**, **coreanos**, **indianos**, **isrealenses** e muitos outros, cada qual com seus caracteres particulares e outros em comum.

Portanto, o Python deve suportar a todos eles, suas particularidades e aquilo que há em comum entre eles.

Edentação

Outra característica fundamental do Python, é que ele foi pensado **para ser compreendido** pelo maior número possível de pessoas. Isso fez com que algumas definições *milenares* fossem chutadas para o espaço em busca de algo prático, claro e que ajudasse tudo isso de forma simpática (neste momento, **Ballzer** soltou um sorriso tão largo que quase separou seu corpo em dois). A grande sacada foi a **edentação** padronizada para todo o código-fonte.

A edentação da linha muda quando esta faz parte de um **bloco** declarado na linha anterior.

Blocos são sempre iniciados por uma linha que termina com o sinal de **dois-pontos** (**:**).

Veja este código novamente, por exemplo:

```
# -*- coding: utf-8 -*-


def imprimir_duas_vezes(texto):
    # Imprime o texto uma vez
    print texto


    # É aqui que imprime outra vez
    print texto


imprimir_duas_vezes('Tarsila do Amaral')
```

O Python segue as linhas do arquivo de cima para baixo até encontrar a seguinte linha:

```
def imprimir_duas_vezes(texto):
```

Esta linha define uma nova função, e inicia o bloco de código que **faz parte** dessa nova função. Daí em diante, as linhas são edentadas com **4 espaços** da esquerda para a direita, o que dá uma leitura agradável do código para quem estava lá fora tomando um ar e chegou ali, coçando a orelha direita, enquanto mastiga um palito de dentes ainda do almoço.

A súbita sensação é esta:

- Sim, claro, já entendi o que esse código faz.

As linhas continuam edentadas até que uma delas volta à edentação anterior, veja:

```
imprimir_duas_vezes('Tarsila do Amaral')
```

Ou seja, **nenhum espaço à esquerda**. Isso quer dizer que aquele bloco acabou, aquele trecho de código da função durou enquanto a edentação estava lá, com **4 espaços** ou mais à esquerda, mas agora que a edentação voltou ao estado inicial, *voltamos* ao assunto anterior.

Compreendido?

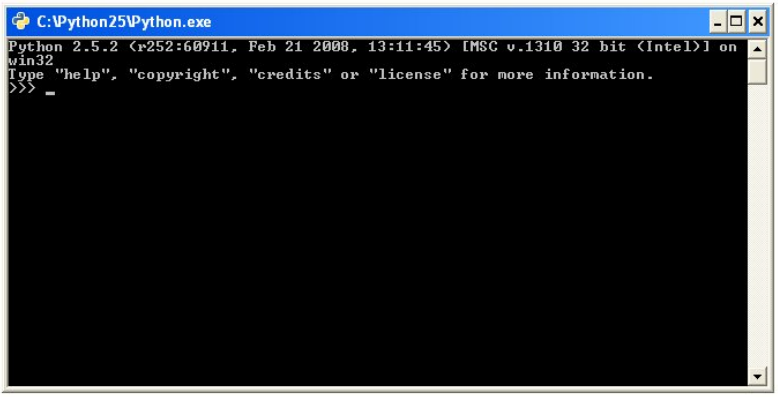
Use sempre múltiplos de **4 espaços** (4, 8, 12, etc.). Nunca utilize **tabulação**, pois isso seria desastroso para o cara do palito de dentes, para o seu substituto nas férias ou para você mesmo, alguns dias depois. **Use sempre 4 espaços para edentar de bloco em bloco**.

Conhecendo o Python interativo

Agora, vá ao **Menu Iniciar** e escolha a opção **"Executar Programa"**. Escreva "python" no campo e clique em **"Ok"**.

No Linux, você pode executar o **Console**, digitar **"python"** e teclar ENTER.

A janela de modo texto do **MS-DOS** (ou do **Console**, no Linux) será aberta com um prompt assim:



Pois bem, este é o **Python interativo**. É importante conhecê-lo, pois constantemente fazemos tarefas de administração de projetos usando esta ferramenta, que é muito prática também para o aprendizado.

Tipagem

No Python, a tipagem dos dados e objetos é **dinâmica** e **forte**.

"Dinâmico" quer dizer que ao uma nova variável, ou um parâmetro de uma função, não é preciso informar qual é o seu tipo, nem mesmo defini-la antes de usá-la. Basta atribuir um valor a ela quando necessário.

No entanto, internamente todo valor é baseado em algum **tipo de dado**, que por sua vez é uma **classe**. Uma vez que um valor é atribuído a uma variável, ela passa a se comportar da forma que a classe daquele valor foi definida para se comportar. Por isso, ela é **"forte"**.

Para ver como isso funciona, volte ao **shell interativo** e digite a sequência de comandos abaixo. Escreva somente as linhas iniciadas por **">>> "**, e claro, ignore o **">>> "**, digitando somente o restante da linha.

```
>>> nome = 'Leticia'

>>> nome
```

```
'Leticia'
>>> nome.__class__
<type 'str'>
>>> type(nome)
<type 'str'>
```

Nas linhas de código acima, nós fizemos o seguinte:

1. declaramos uma variável **"nome"** com uma *string* **'Leticia'**;

2. escrevemos o nome da variável para que seu valor original seja exibido na linha de baixo, ou seja: **'Leticia'**;

3. na linha seguinte, informamos o atributo interno do Python que retorna a classe daquele objeto, e ele mostrou que o valor contido na variável **"nome"** é um tipo básico de string (**"<type 'str'>"**);

4. e na última linha usamos uma outra forma de fazer a mesma coisa da linha anterior.

Agora faça algo semelhante, só que desta forma:

```
>>> idade = 28
>>> idade
28
>>> idade.__class__
<type 'int'>
```

Veja que o valor **"28"** foi interpretado como número inteiro (**"<type 'int'>"**).

Veja esses outros:

```
>>> nota = 8.7
>>> nota
8.6999999999999993
>>> nota.__class__
<type 'float'>
>>> mulher = True
>>> mulher
True
>>> mulher.__class__
<type 'bool'>
>>> nota * idade
```

```
243.59999999999997
>>> idade * nota
243.59999999999997
```

Import explícito e namespaces

Agora, depois de ver alguns **tipos de dados básicos**, vamos partir para outro um pouquinho mais complexo. No shell interativo, digite a seguinte sequência de comandos:

```
>>> from datetime import date
>>> hoje = date.today()
>>> hoje
datetime.date(2008, 11, 24)
>>> hoje.__class__
<type 'datetime.date'>
```

Como pode ver, para trabalhar com datas, é preciso **importar** um pacote chamado **"date"**, de dentro de outro pacote, chamado **"datetime"**.

Importar um pacote ou objeto trata-se de fazer o Python saber que quando aquele elemento for citado, ele deve ser localizado em uma biblioteca diferente do módulo em que você está agora, e diferente também da biblioteca *built-in* do Python, que é carregada automaticamente.

Há outras formas de se importar um pacote ou objeto, veja:

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2008, 11, 24)
```

Observe que aqui, nós importamos o pacote **"datetime"** inteiro. Então, para usá-lo, precisamos informar todo o caminho para a função **today()**.

```
>>> datetime = __import__('datetime')
>>> datetime.date.today()
datetime.date(2008, 11, 24)
```

Neste outro bloco, importamos o pacote **"datetime"** da mesma forma, só que agora seu nome foi informado dentro de uma string. Isso é útil para se importar pacotes quando seus nomes serão conhecidos somente em tempo de execução.

```
>>> from datetime import *
>>> date.today()
datetime.date(2008, 11, 24)
```

Já neste último bloco, importamos todos os elementos de **"datetime"**, o que inclui também o elemento **"date"**, então o usamos logo em seguida.

Acontece, que Alatazan sabe que quando se traz uma manada de elefantes de uma região da floresta, e outra manada de elefantes de outra região da floresta, mas não se usa nem um papelzinho para dizer que onde vieram, então a bagunça está definitivamente... **feita**!

E quando se usa esse famigerado **"from algum_lugar import *"** e outro **"from outro_lugar import *"**, acontece a mesma coisa: você não sabe quem veio, e também não sabe de onde um elemento específico veio, o que é muito ruim.

Portanto não é aconselhável o uso de **"import *"**, a menos que seja em um caso específico e simples.

Isso se chama **"namespace"**.

Declaração de classes

Vamos agora **declarar uma classe** no shell? Pois então digite as linhas de código abaixo. As linhas iniciadas com **"... "** são blocos, portanto, use os **4 espaços** para edentar o bloco.

```
>>> class Pessoa:
...     nome = 'Leticia'
...     idade = 28
...     nota = 8.7
...     mulher = True
...     # nao se esqueca da edentacao para continuar o bloco
...     def andar(self, passos=10):
...         if self.idade > 120:
...             print 'Nao foi possivel'
...         else:
...             print 'Andou %d passos!'%passos
...
>>>
```

Puxa, fizemos um bocado de coisas aí.

Primeiro, você deve ter notado que a cada início de um novo bloco, mais 4 espaços devem acrescentar à edentação e o contrário também é verdadeiro.

Em segundo, declaramos alguns atributos para a classe (**nome**, **idade**, **nota** e **mulher**) já com valores iniciais. Isso não é obrigatório, mas dá uma **clareza**

aliviadora para **aquele mesmo cara do palito**.

Em terceiro, você deve notar que, no shell interativo, quando um bloco tem um espaço em branco entre suas linhas (para manter a estética e facilidade de leitura), este deve seguir a mesma edentação, justamente porque senão ele vai pensar que você acabou de fechar o bloco, o que não é a sua intenção naquele momento. Mas essa preocupação não é necessária quando se está escrevendo código em um arquivo **.py**.

Self explícito na declaração de métodos

Por último, declaramos o método **andar(self, passos=10)**. Para ser um método é preciso ter **self** como primeiro parâmetro e ser uma função. O parâmetro **passos** possui um valor *default*, ou seja, caso não seja especificado, ele assumirá o valor **10**.

Agora escreva as seguintes linhas de código na sequência:

```
>>> mychell = Pessoa()
>>> mychell.nome = 'Mychell'
>>> mychell.idade = 15
>>> mychell.nome
'Mychell'
>>> mychell.idade
15
>>> mychell.andar()
Andou 10 passos!
>>> mychell.andar(15)
Andou 15 passos!
```

Primeiro, instanciamos a classe **Pessoa** e atribuímos essa instância à variável **"mychell"**.

Depois atribuímos valores a seus atributos, para modificar os valores iniciais.

E por último, executamos o método **andar**.

Na primeira execução do método, não informamos um valor para o parâmetro **passos**, então o valor assumido foi **10**. Veja:

```
>>> mychell.andar()
Andou 10 passos!
```

Na segunda execução, informamos um valor para o parâmetro. Observe que o parâmetro **self** foi completamente ignorado. Isso é porquê ele deve ser informado

somente na declaração, e não na execução do método.

```
>>> mychell.andar(15)
Andou 15 passos!
```

Atribuição de método e inicializador de classe

Pois agora vamos brincar um pouco com programação **funcional** e atribuir um método especial de **inicialização** à classe **Pessoa**, veja:

```
>>> def inicializador(self, nome, idade, nota=None, mulher=None):
...     self.nome = nome
...     self.idade = idade
...     self.nota = nota or self.nota
...     self.mulher = mulher is not None or self.mulher
...
>>> Pessoa.__init__ = inicializador
```

Veja só:

Primeiro, declaramos uma nova função, chamada **"inicializador"**, com os atributos **self**, **nome**, **idade**, **nota** e **mulher**. Os dois últimos possuem valores *default*.

A seguinte linha diz, em outras palavras: "o atributo **"self.nota"** deve receber o valor do parâmetro **"nota"** somente se este tiver um **valor válido**, do contrário, recebe o seu próprio valor para continuar tudo como está".

Neste caso, um **valor válido** é um valor **verdadeiro**. Para o Python, os valores **0**, **False**, **None**, **''**, **()**, **[]** e **{}** não são verdadeiros.

```
...     self.nota = nota or self.nota
```

Já na linha abaixo, a coisa é bem parecida, mas o valor do parâmetro **"mulher"** só será atribuído ao atributo **"self.mulher"** caso ele **não seja None**, ou seja, seu valor *default*.

```
...     self.mulher = mulher is not None or self.mulher
```

É importante perceber aqui, que o que estamos fazendo é atribuir os valores informados para os parâmetros **opcionais** somente se estes foram realmente informados, pois **None** é seu valor default, e indica que **ele não foi informado**.

Na segunda parte, atribuímos a função à classe, como se fosse um atributo, de nome **__init__**, mas como ela é uma função, o Python a atribui como **método**.

Agora veja a seguir:

```
>>> ana = Pessoa()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: inicializador() takes at least 3 arguments (1 given)
```

No código acima, ao instanciar a classe **Pessoa**, desta vez não foi possível, pois o seu **inicializador** obriga que pelo menos os parâmetros **sem valor** *default* sejam informados.

```
>>> ana = Pessoa('Ana Paula', 8)
```

Agora fizemos do jeito certo, atribuindo o **nome 'Ana Paula'** e a **idade 8**.

Ao imprimir os valores dos atributos na tela, eles são exibidos, até mesmo o atributo **"mulher"**, que permaneceu com seu valor *default*.

```
>>> ana.nome
'Ana Paula'
>>> ana.idade
8
>>> ana.mulher
True
```

Para finalizar essa sequência, você deve saber que poderia informar o **inicializador** com qualquer outro nome, ou poderia - o que é o mais indicado sempre - simplesmente declarar o método da forma convencional, assim por exemplo:

```
class Pessoa:
    nome = 'Leticia'
    idade = 28
    nota = 8.7
    mulher = True

    def __init__(self, nome, idade, nota=None, mulher=None):
        self.nome = nome
        self.idade = idade
        self.nota = nota or self.nota
        self.mulher = mulher is not None or self.mulher

    def andar(self, passos=10):
        if self.idade > 120:
            print 'Nao foi possivel'
```

```
        else:
            print 'Andou %d passos!'%passos
```

O resto também deve ser explícito

O Python segue a ideia de que tudo deve ser posto às caras, livre, e muito, muito explícito. Use nomes de variáveis, classes e funções, todos explícitos. Comentários explícitos, etc... tudo deve ser claro... e falando nessas coisas, também é recomendável...

Padrão de nomenclatura

... que as variáveis tenham seus nomes em caixa baixa (letras minúsculas), com as palavras separadas por *underscore* (_), assim:

```
nome = 'Marta'
data_nascimento = None
```

... que funções e métodos sejam verbos, assim:

```
def andar_para_frente():
    print nome, 'andou'
```

... e que as classes tenham nomes no singular, com a primeira letra de cada palavra em caixa alta, assim:

```
class PessoaFisica:
    nome = 'Leticia'
```

Comentários

... e que comentários podem ser feitos assim:

```
# linha comentada que sera ignorada da mesma forma
# que esta outra linha
```

ou assim:

```
"""linha comentada que sera ignorada da mesma forma
que esta outra linha"""
```

E é muito aconselhável que todas as funções e classes tenham um comentário no início do bloco, assim:

```
class Pessoa:
    """Classe para atribuicao a pessoas do sistema"""
    nome = None
```

```
    idade = 20
```

Pois ao chamar a função **help()** para aquela classe ou função no shell interativo, aquele comentário será interpretado como a **documentação** da classe. Veja com seus próprios olhos:

```
>>> help(Pessoa)
Help on class Pessoa in module __main__:


class Pessoa
 |  Classe para atribuicao a pessoas do sistema
 |
 |  Data and other attributes defined here:
 |
 |  nome = None
 |  idade = 20
```

Não é fantástico?

Funções dir() e help()

Aliás, falando em função **help()**, ela é uma forma muito bacana de aprender Python, veja:

```
>>> import datetime
>>> help(datetime)
Help on built-in module datetime:


NAME
    datetime - Fast implementation of the datetime type.


FILE
    (built-in)


CLASSES
    __builtin__.object
        date
            datetime
        time
```

```
---- cortamos aqui pois a documentação deste pacote é extensa
```

Ou este:

```
>>> import urllib
>>> help(urllib)
Help on module urllib:


NAME
    urllib - Open an arbitrary URL.


FILE
    c:\python25\lib\urllib.py


DESCRIPTION
    See the following document for more info on URLs:
    "Names and Addresses, URIs, URLs, URNs, URCs", at
    http://www.w3.org/pub/WWW/Addressing/Overview.html


    See also the HTTP spec (from which the error codes are
derived):
    "HTTP - Hypertext Transfer Protocol", at


---- cortamos denovo
```

Isso é muito útil e ajuda bastante.

E tão útil quanto a função **help()** é a função **dir()**, veja

```
>>> dir(Pessoa)
['__doc__', '__init__', '__module__', 'andar', 'idade', 'mulher',
'nome', 'nota']
```

A função **dir()** retorna em uma lista todos atributos e métodos daquela instância, e isso vale também para classes.

Portanto, se você quer saber quais atributos e métodos a string **'Beatles'** possui, faça isto:

```
>>> dir('Beatles')
[
```

```
'__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__',
'__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize',
'center', 'count', 'decode','encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill'
]
```

E agora, sabendo dos atributos e métodos dessa string, volte ao **help()**:

```
>>> help('Beatles'.upper)
Help on built-in function upper:


upper(...)
    S.upper() -> string


    Return a copy of the string S converted to uppercase.
```

Novamente: **o Python é fantástico**!

Com muita empolgação, rumo ao *deploy*

Alatazan ficou fascinado, e Ballzer, como era de se esperar, começou a formular suas aventuras futuras.

- Puxa, evidentemente o Python é mais do que isso, mas só em saber disso muito do que já fizemos e vamos fazer nos próximos dias se esclareceu de forma tão... *pythonica*!

- Meu camarada, é muito, muito bacana programar em Python. E o mais interessante é que as coisas são realmente simples, veja que é muito fácil casar classes com funções de formas variadas. É por isso que existem tantos *frameworks* criados em Python!

Cartola ajustou o volume de seu iPod e prosseguiu:

- Vamos resumir a parada:
    - Python é multi-paradigma: **orientado a objetos, orientado a aspectos, procedural, funcional e outros**;
    - pode ser compilado ou interpretado;
    - roda em qualquer sistema operacional que seja popular;
    - Python é unicode, por isso, é preciso informar a codifição do arquivo se for usar caracteres especiais;
    - Python constrói blocos por edentação, sempre de 4 em **4 espaços**;
    - O **shell interativo** é uma mão-na-roda em muitas situações;
    - a tipagem de dados no Python é **dinâmica e forte**;
    - faça tudo **explícito** e use *namespaces*;
    - classes também são objetos, e métodos devem ser declarados sempre com **self** sendo o primeiro parâmetro;
    - funções comuns podem ser transformadas em métodos;
    - comentários em string no início do bloco são a **documentação** daquela função ou classe;
    - as funções **help()** e **dir()** abrem as cortinas para o esclarecimento.

Pronto! Agora com conceitos básicos sobre HTML, CSS e Python, podemos seguir adiante. No próximo capítulo, vamos fechar a primeira iteração e partir para mais aventuras!

Capítulo 14: Ajustando as coisas para colocar no ar

Nesse dia, caiu uma chuva daquelas que não têm nada a ver com o tempo nem com a época.

Alatazan nem sabia disso, pois estava no planeta há poucas semanas. Mas ele sentiu um gostinho de vingança quando viu os panfletos de gás, supermercado, mãe-de-santo e outros mais derretendo na caixinha de correios.

Definitivamente: se havia uma coisa chatíssima na Terra eram esses malditos panfletos.

Alatazan compreendeu mais rápido que ninguém a ideia de **"colocar no ar"**. Em Katara, há empresas de publicidade que colocam milhares de propagandas no ar todos os dias.

O problema é que essas propagandas são pequenos **robôs** que se parecem com **moscas**, e ficam em grupos de 5 ou 6 por ali, perto dos portões como se não quisessem nada, assobiando e praticando um voo manjado. Quando o infeliz morador sai de sua casa, é atacado por eles com as mais variadas propagandas. Às vezes essas pessoas entram em ataque de nervos e passam um bom tempo tentando acertar os robôs com tapas ou jornais, mas nunca se resolve, e no outro dia é a mesma coisa.

O pai de Alatazan trabalha na fábrica onde esses robôs são feitos, na seção onde eles são treinados, e ali eles usam uma capa frágil, resistente a coisa nenhuma. Depois do treinamento eles recebem uma capa resistente e flexível, que não permite que se derretam quando a chuva cai.

Só depois disso, podem ser *colocados no ar*.

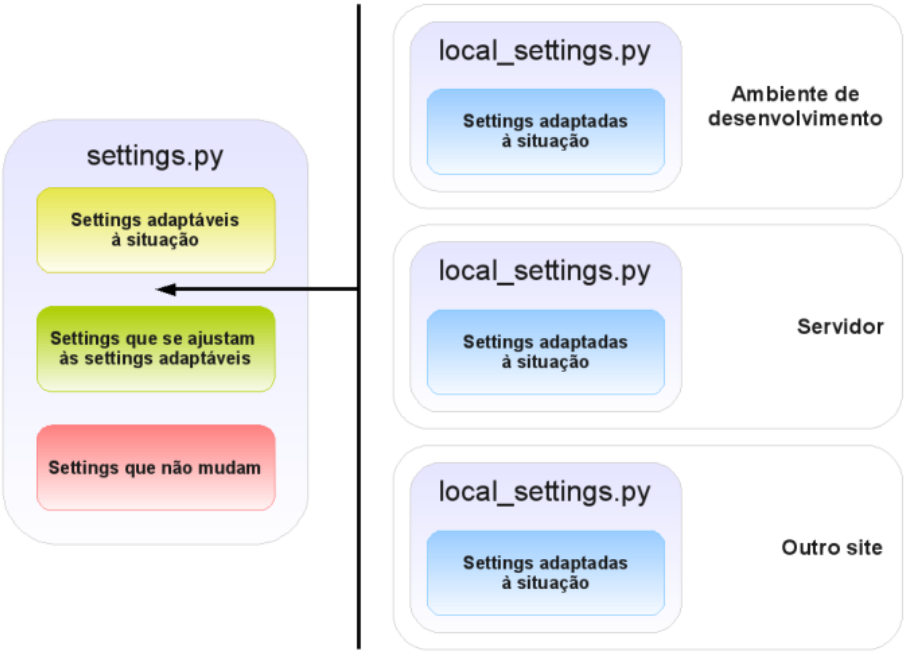
Vamos agora preparar o projeto

Para enviar o projeto a um servidor e colocá-lo no ar, é importante fazer algumas modificações para isso. No servidor, ele deve se comportar de forma mais leve, sem recursos de **depuração** e outras coisinhas.

Então, mão na massa!

Antes de mais nada, execute seu projeto clicando duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto.

Primeiro de tudo, abra o arquivo **"settings.py"** da pasta do projeto para edição. Vamos pensar as **settings** do projeto na seguinte divisão:



Há algumas **settings** que são muito relativas ao ambiente ao qual seu projeto está no momento. Vamos dar o exemplo mais simples: o estado de **DEBUG**. Esta setting só deve ter valor **True** em ambientes de **teste ou desenvolvimento**, nunca em produção.

Outras settings já são resultado de uma definição fixa, somada a outra setting do caso citado acima. Um bom exemplo é a setting **MEDIA_ROOT** que seria sempre a mesma pasta **"media"** da pasta do projeto, mas a pasta do projeto pode estar em um caminho no ambiente de desenvolvimento e em outro no servidor.

E há ainda as settings que dificilmente mudam, como **INSTALLED_APPS** por exemplo.

A definição de quais settings são flexíveis e quais não o são variam de acordo com o projeto, mas em geral há um ponto divisor no arquivo **"settings.py"** que se adapta à maioria dos casos.

Localize a seguinte linha no arquivo **"settings.py"**:

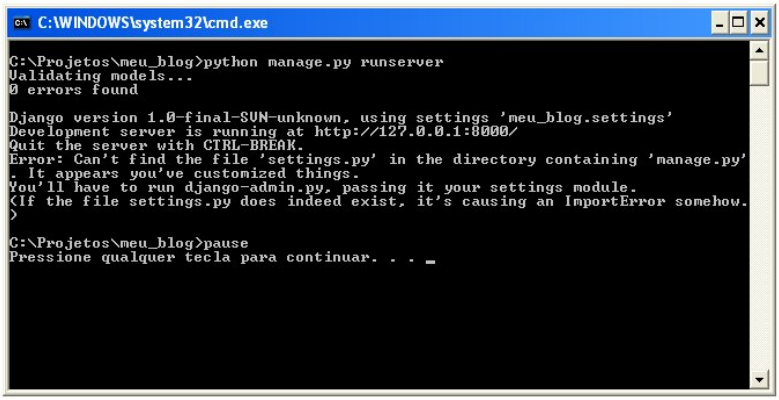
```
USE_I18N = True
```

Na maioria dos projetos, esta é a última linha de um grupo de settings que variam muito de acordo com a situação atual do projeto. Abaixo desta linha estão aquelas settings que se ajustam a elas, ou que não se ajustam a nada.

Portanto, vamos acrescentar um novo trecho de código abaixo dessa linha:

```
from local_settings import *
```

Salve o arquivo e vá à janela do **MS-DOS** (ou **Console**, no caso do Linux) onde o projeto está rodando. Veja o que aconteceu:



O erro que surgiu é este:

```
Error: Can't find the file 'settings.py' in the directory
containing 'manage.py'.

It appears you've customized things.

You'll have to run django-admin.py, passing it your settings
module.

(If the file settings.py does indeed exist, it's causing an
ImportError somehow.)
```

Ou seja, nosso projeto caiu e parou de funcionar, porque houve um erro do tipo **ImportError** no arquivo **"settings.py"**. Isso aconteceu porque o arquivo importado **"local_settings.py"** não foi encontrado.

Pois agora volte ao arquivo **"settings.py"** e modifique o trecho de código que você acrescentou para ficar assim:

```
try:

    from local_settings import *
```

```
except ImportError:
    pass
```

Salve o arquivo. Execute o projeto novamente, clicando duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto.

No Python, o **try/except** é um recurso que permite que você faça uma tentativa, e caso essa tentativa resulte em erro, você pode tratar esse erro sem que seu software seja afetado ou paralisado.

Em outras palavras nós tentamos importar todas as settings do arquivo **"local_settings.py"**, mas caso ocorra algum **erro de importação** (ImportError), a situação será ignorada e o interpretador do Python deve seguir adiante como se nada tivesse acontecido. Veja que apenas erros de importação estão sendo suportados aqui, ou seja, qualquer outro tipo de erro terá o comportamento padrão de levantar uma exceção.

O erro de importação mais comum é o de o arquivo não existir.

O raciocínio é o seguinte: todas as settings do arquivo **"local_settings.py"** serão importadas, substituindo o valor das settings que estão acima desse trecho de código. Se elas não tiverem sido declaradas no **"local_settings.py"**, permanecem como estão.

Agora, na pasta do projeto, crie um novo arquivo chamado **"local_settings.py"** com o seguinte código dentro:

```
import os
PROJECT_ROOT_PATH = os.path.dirname(os.path.abspath(__file__))


LOCAL = True
DEBUG = True
TEMPLATE_DEBUG = DEBUG


DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = os.path.join(PROJECT_ROOT_PATH,'meu_blog.db')
```

Estas são as settings que fatalmente serão modificadas quando o site for levado para o servidor, portanto, ao trazê-las para cá, garantimos que na máquina local elas permanecerão como estão.

Salve o arquivo. Feche o arquivo.

Agora de volta ao arquivo **"settings.py"**, localize esta linha:

```
DEBUG = True
```

Modifique-a para ficar assim:

```
DEBUG = False
```

E acima dela, acrescente a seguinte linha:

```
LOCAL = False
```

Ao mudar a setting **DEBUG** para **False**, desativamos o **estado de depuração** do projeto em suas configurações padrão, mas elas permanecerão funcionando localmente, pois no arquivo **"local_settings.py"** elas foram mantidas como estavam.

Ok, por agora isso é suficiente.

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"urls.py"** da pasta do projeto para edição e localize o seguinte techo de código:

```
    (r'^media/(.*)$', 'django.views.static.serve',
     {'document_root': settings.MEDIA_ROOT}),
```

É a nossa URL para arquivos estáticos, certo? **Remova** esse trecho de código.

Agora ao final do arquivo, acrescente as seguintes linhas:

```
if settings.LOCAL:
    urlpatterns += patterns('',
        (r'^media/(.*)$', 'django.views.static.serve',
         {'document_root': settings.MEDIA_ROOT}),
    )
```

O que fizemos aí foi o seguinte: a URL **'^media/'** só deve existir se o projeto estiver em máquina **local**, ou seja, na sua máquina, em ambiente de desenvolvimento.

Isso é importante, pois o Django **não foi feito** para servir arquivos **estáticos** em servidores. No servidor esta tarefa deve ser deixada para um software que foi criado para isso: um **servidor web**, como o **Apache**, Lighttpd, IIS ou outro.

Agora o arquivo **urls.py** ficou assim:

```
from django.conf.urls.defaults import *
from django.conf import settings


# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()
```

```
from blog.models import Artigo
from blog.feeds import UltimosArtigos

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
         'date_field': 'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
    (r'^contato/$', 'views.contato'),
    (r'^comments/', include('django.contrib.comments.urls')),
)

if settings.LOCAL:
    urlpatterns += patterns('',
        (r'^media/(.*)$', 'django.views.static.serve',
         {'document_root': settings.MEDIA_ROOT}),
    )
```

Salve o arquivo. Feche o arquivo.

A divisão que trabalhamos no arquivo **"settings.py"**, que você pôde observar no diagrama, é simples de se fazer e poderosa. Pois seguindo essa linha de raciocínio é possível não só configurar o projeto para diferenciar o **servidor em produção** de seu ambiente de **desenvolvimento**, como é útil também para ambientes de **homologação e testes**, situações de um projeto em **mais de um servidor** ou de **múltiplos sites** em um só projeto.

Tire um tempo para pensar nisso, observe o arquivo **"settings.py"** com atenção e note aquilo que poderia variar em situações distintas. Se for necessário, mova a setting desejada para antes ou depois do trecho de código que importa o módulo **"local_settings.py"**.

Agora, para concluir, abra novamente o arquivo **"settings.py"** do projeto e localize a seguinte linha:

```
ROOT_URLCONF = 'meu_blog.urls'
```

Agora a modifique para ficar assim:

```
ROOT_URLCONF = 'urls'
```

Esta modificação é recomendável, pois assim seu projeto trabalha de uma forma transparente em relação à **PYTHONPATH** - o conjunto de pastas reconhecidos pelo Python em sua máquina virtual para encontrar pacotes e módulos. Desta forma, quando seu projeto for lançado ao servidor, vamos adicionar a pasta do projeto à **PYTHONPATH** e tudo estará resolvido.

Salve o arquivo. Feche o arquivo.

Pronto. Os primeiros passos para levar o projeto ao servidor são esses.

Vamos adiante?

Alatazan perguntou, só pra confirmar:

• Então o projeto tem comportamentos diferentes, dependendo se está no servidor ou em minha máquina local?

Ao que Cartola respondeu:

• Isso, e há um grupo de settings potencialmente adaptáveis a essas situações diferentes, relacionadas a caminhos de pastas, modo de depuração e outras coisas assim...

• E tem também aquelas que são flexíveis, pois recebem valores somados a valores de settings modificadas em linhas de código anteriores ou no arquivo **"local_settings.py"**.

• Por último, há aquelas que dificilmente vão se modificar, pois estão na estrutura do projeto, como **MIDDLEWARE_CLASSES** ou **INSTALLED_APPS** por exemplo.

• Lembrando que o arquivo **"settings.py"** deve permanecer com as configurações **ideais** para o projeto, ou seja, as que serão usadas caso o arquivo **"local_settings.py"** não exista.

• O que você precisa ficar ligado é que, ao levar o seu projeto para o servidor, o arquivo **"settings.py"** deverá permanecer idêntico, sem alterações, pois elas devem ser feitas **apenas** no arquivo **"local_settings.py"**. Se ao levar seu projeto para o servidor, você fizer mudanças em outras partes do código, é muito provável que esteja no caminho errado. **Não se esqueça disso!**

O bate-bola de Cartola e Nena resumiu rapidamente o aprendizado do dia, e Alatazan notou aqui uma diferença fundamental de hoje para os dias anteriores: agora estavam entrando em uma fase mais **profissional** da coisa.

Capítulo 15: Infinitas formas de se fazer deploy

Próximo a onde Alatazan viveu sua infância há uma cidade chamada **Konnibagga**, que foi construída dentro de uma ilha, **no meio de um lago**.

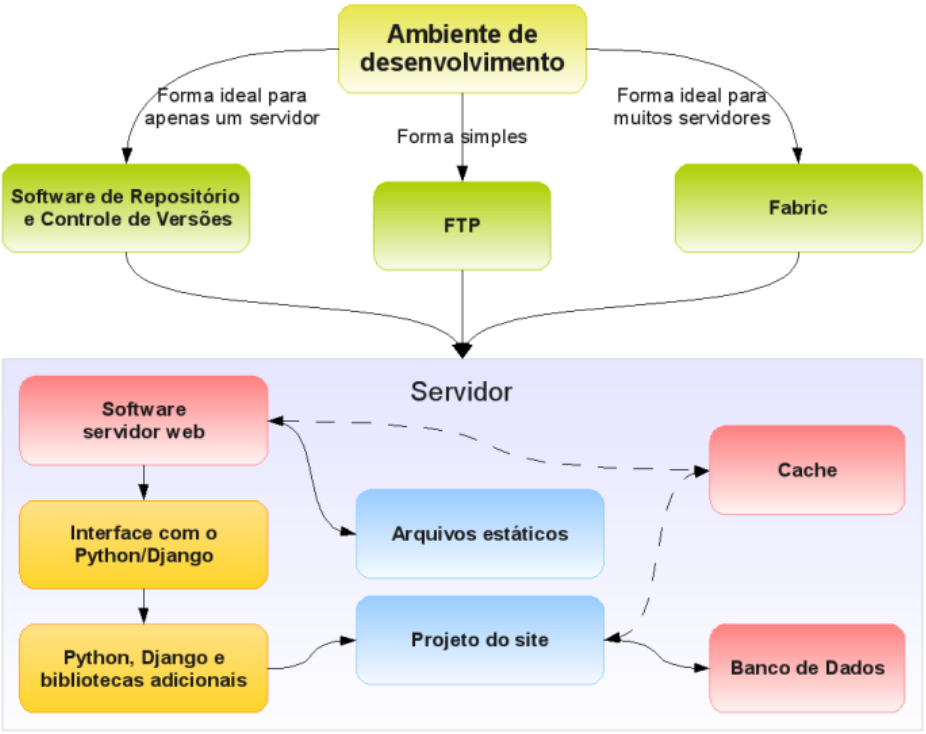
Devido à sua população ser extremamente **dinâmica**, através dos séculos **dezenas de pontes** foram construídas para atravessar o lago **até a ilha**. Haviam pontes compridas, pontes curtas, pontes gratuitas, pontes com pedágio, pontes bonitas e feias, resistentes e frágeis, largas e estreitas.

Um dia um homem sugeriu que se fossem atravessadas todas as pontes em sequência, entrando e saindo da cidade, seria possível desenhar um girassol com o traçado do percurso. Acontece que em Katara não existem girassóis e ninguém compreendeu qual era o fundamento para tal sugestão.

O homem foi então contratado pela prefeitura para construir mais pontes por um lucro maior desde que parasse de falar nessa ideia, e assim ele realmente parou com isso, construiu muitas pontes e com o dinheiro fez uma viagem transatlântica - apesar de lá também não haver um Atlântico para atravessar.

As muitas maneiras de se colocar um site no ar

Há diversos caminhos - talvez infinitos - para se colocar um site no ar. Porque para isso é necessário combinar uma série de passos que todos eles possuem diversas alternativas.



Na ilustração acima estão dispostos os elementos de um deploy básico. Para cada um desses passos há alternativas variadas para escolher, dependendo da situação e preferência dos desenvolvedores e serviço de hospedagem.

As caixas em verde (**Software de Repositório e Controle de Versões**, **FTP** e **Fabric**) são caminhos os quais você vai usar para levar os arquivos do projeto para o servidor. Cada um deles se adequa a uma realidade diferente.

As caixas em rosa (**Software servidor web**, **Cache** e **Banco de Dados**), são softwares que normalmente são instalados pelo serviço de hospedagem em seu servidor, portanto, caso contrate um serviço de hospedagem é muito provável que esses softwares já vêm instalados e configurados.

As caixas em laranja (**Interface com o Python/Django** e **Python, Django e bibliotecas adicionais**) são softwares que (exceto pelo Python em servidores Linux) normalmente não vêm instalados nos servidores e provavelmente terão de ser instalados e configurados por você mesmo.

E as caixas em azul (**Arquivos estáticos** e **Projeto do site**) são os arquivos de seu projeto. Portanto, pelo menos esses arquivos você terá de levar para o servidor para ter seu site no ar.

Ambiente de desenvolvimento

Seu ambiente de desenvolvimento (ou de sua equipe) deve estar preparado para fazer testes e desenvolvimento local e uma ferramenta para enviar seus arquivos para o servidor, que pode ser de um dos 3 tipos a seguir.

Software de Repositório e Controle de Versões

Quando se trabalha com o projeto em apenas um servidor, especialmente quando há mais de uma pessoa envolvida no desenvolvimento do site, é recomendado que se use uma ferramenta de **repositório e controle de versões**.

Os exemplos mais populares desse tipo de software são:

- Git
- Subversion
- Bazaar
- Mercurial

A recomendação para a maioria dos casos é o **Git**, atualmente a ferramenta com maior flexibilidade e performance dentre as conhecidas no mercado. E além disso, é livre e gratuita.

FTP

O uso do FTP tem se tornado cada vez menos recorrente devido à falta de controle e dificuldade para se trabalhar em equipe. Ainda assim, é a ferramenta mais simples e presente nos servidores, conhecida pela maior parte dos desenvolvedores.

O fato é que de uma ou outra forma, nunca se deixa de usar o FTP, mas seu uso passa a ser para situações de casos isolados ou emergenciais.

Fabric

Por último, a mais poderosa opção para fazer deploy de projetos em Django se chama **Fabric**. Você pode conseguir mais detalhes sobre ele em seu site oficial:

```
http://www.nongnu.org/fab/
```

Seu uso se resume a construir um pequeno script, chamado de **fabfile**, que determina as regras para se atualizar servidores em grande número, realizando algumas tarefas relacionadas a isso. Não se trata de uma opção vantajosa para a maioria do sites que estão presentes em somente um servidor, mas para ambientes mais complexos se trata de uma grande vantagem.

Software servidor web

Os servidores web se tratam de softwares que *escutam* por uma porta **HTTP** ou **HTTPS** (geralmente portas **80** e **443**, respectivamente) e servem aos navegadores com arquivos e conteúdo dinâmico. Trata-se de um dos elementos mais fundamentais neste processo. É impossível evitar o uso de algum tipo de servidor.

Mas há diversos tipos, versões e formas de configurá-los. Os mais populares são:

- Apache
- Nginx
- Lighttpd
- Microsoft IIS

Normalmente as diferenças mais comuns que se atribui entre esses servidores são:

- **Apache** é o mais robusto, mais popular e mais compatível deles, especialmente quando combinado com outros softwares livres.
- **Nginx** é relativamente novo, é muito leve e rápido e trabalha com um cache próprio de conteúdo. No entanto é escasso de documentação.
- **Lighttpd** é excelente para servir arquivos estáticos e FastCGI, possui uma forma de configuração bastante flexível e é também muito leve.
- **Microsoft IIS** é o mais compatível com tecnologias da linha Microsoft. Entretanto é o mais limitado quando se refere à combinação com softwares livres, possui uma estrutura fechada e diversos problemas de segurança.

Cada um deles possui uma forma diferente de se configurar para trabalhar com o **Django**. No próximo capítulo vamos trabalhar a configuração do Apache, o mais popular e comum deles.

Banco de Dados

Existem diversos bancos de dados. O Django é compatível oficialmente com 5:

- MySQL
- PostgreSQL
- SQLite
- Oracle
- Microsoft SQL Server

Os primeiros três são livres, sendo que o MySQL e PostgreSQL são mais indicados para servidores em produção, pois são poderosos e possuem uma performance relativamente superior aos demais. Além disso são plenamente compatíveis com outros softwares livres.

O SQLite é mais indicado para ambientes de desenvolvimento, por ser muito simples e leve, mas não possui uma série de recursos desejáveis para servidores.

Já o Oracle e MS SQL Server são softwares proprietários, que dispõem de versões gratuitas mas limitadas. Também são menos compatíveis com softwares livres.

No próximo capítulo vamos trabalhar na configuração de um servidor com MySQL.

Cache

O serviço de Cache em um servidor pode variar muito. Muitos servidores já são oferecidos por empresas de hospedagem com um software de cache, como os seguintes:

- Squid
- Varnish
- memcached

Ainda que figurem como ferramentas para cache, os três softwares citados acima possuem características e aplicações diferentes.

Há ainda outras formas de fazer cache no Django, usando **banco de dados** ou uma **pasta do HD**. Mas vamos trabalhar a instalação do próximo capítulo com **memcached**, a mais indicada para esse caso.

Interface com o Python/Django

Para que o servidor web se comunique com o seu projeto em Django, é preciso ter uma interface, que possibilita ao servidor o **reconhecimento** de software escrito em Python, que é o caso de projetos em Django.

As formas mais comuns de se fazer isso são:

- mod_python
- WSGI
- FastCGI

Cada um deles possui um método de trabalho diferente, e sua aplicação varia de

acordo com o caso.

A maior parte dos serviços de hospedagem **compartilhada** (quando um mesmo servidor é compartilhado por vários clientes) é feita usando **FastCGI**, mas o crescimento do WSGI nesse nicho tem sido uma constante. Neste caso, usa-se o módulo **mod_rewrite** do Apache ou funcionalidades semelhantes em outros servidores web.

Já a maior parte dos **servidores dedicados** e **servidores virtualizados** são trabalhados usando **mod_python**, mas da mesma forma, o crescimento do **WSGI** tem substituído o mod_python pouco a pouco.

Isso se deve à facilidade e flexibilidade que o WSGI oferece e ao ganho de performance que também oferece na maior parte dos casos.

Python, Django e bibliotecas adicionais

Na maioria das distribuições **Linux**, o **Python** é instalado como parte oficial do sistema operacional. Mas em outros casos é preciso ser feita sua instalação.

É necessário também **instalar o Django**. Não é necessário que ele seja instalado entre os pacotes oficiais do Python, pois pode simplesmente ser colocado na mesma pasta do projeto, mas ainda assim, ele é **necessário**.

É preciso também instalar a **biblioteca de acesso ao banco de dados**. É ela quem faz a ponte entre o Python e o seu banco de dados escolhido.

As bibliotecas de acesso a bancos de dados são estas:

- MySQLdb (para acesso a MySQL)
- pySQLite (para acesso a SQLite)
- cx_Oracle (para acesso a Oracle)
- psycopg (para acesso a PostreSQL)
- psycopg2 (para acesso a PostreSQL)
- PyWin32 (para acesso a Microsoft SQL Server)

Arquivos estáticos

Os arquivos estáticos são arquivos de **imagens, CSS, JavaScript, Flash Shockwaves** e outros. Não é comum lançar um site sem alguns deles.

Quando se faz o *deploy* de um projeto, é preciso configurar o servidor web para publicar seus arquivos estáticos de forma independente do Django. O Django não foi construído para servir arquivos estáticos e não vale a pena perder performance

com isso.

Outra recomendação importante é a **separação** do servidor web do **projeto** do servidor web de arquivos estáticos. Não é necessária a separação entre máquinas diferentes, o que recomendamos é que as configurações dos servidores web sejam independentes uma da outra.

A cada vez que uma nova pessoa entra em seu site, uma máquina virtual Python é instanciada para ela, e isso geralmente custa em torno de 15Mb a 30Mb de memória. Arquivos estáticos não precisam nem de um décimo disso. Se o seu servidor web for configurado para abrir instâncias com pouca memória, sua máquina virtual Python será prejudicada, e caso seja configurado para ocupar muita memória, os arquivos estáticos serão os prejudicados.

Portanto, sempre que possível, crie dois domínios separados, assim por exemplo: para um site que possui o seguinte domínio:

```
http://blog-do-alatazan.com/
```

Tenha um subdomínio assim para arquivos estáticos:

```
http://media.blog-do-alatazan.com/
```

Dessa forma você pode configurar o servidor web de formas diferentes, otimizando cada um de acordo com seu caso. Você pode inclusive separar para softwares servidores web diferentes se preferir. Exemplo: **Apache** para o projeto em Django e **Lighttpd** para arquivos estáticos.

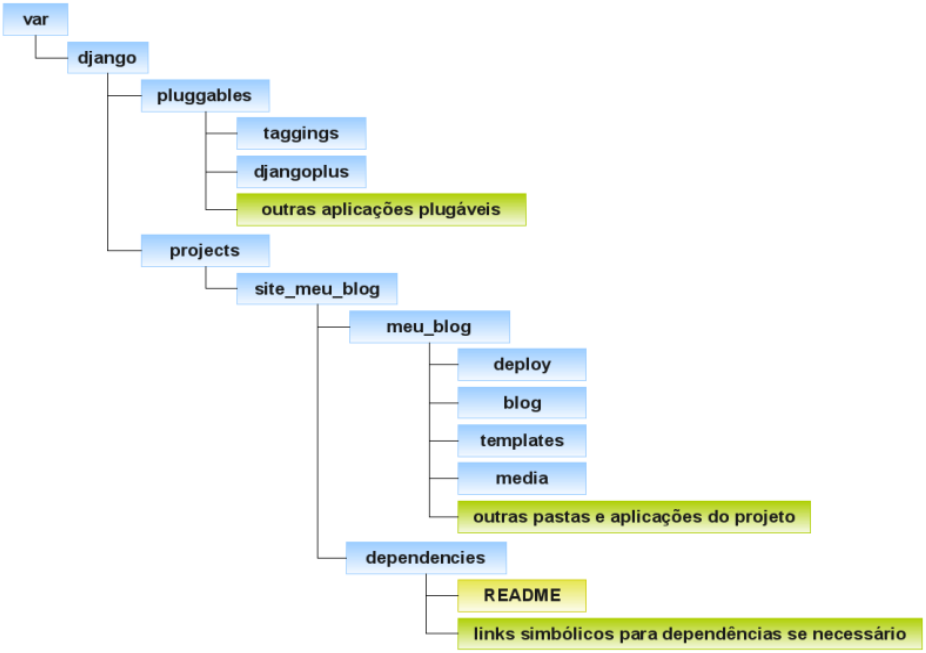
Projeto do site

O projeto é o coração de seu site. Construído em Django, é a única peça de fato específica para o nosso framework. Isso quer dizer que todo o restante do site pode ser aproveitado da mesma forma para **diferentes linguagens e diferentes frameworks**, mas esta é específica para o Django: **seu projeto** e a **configuração** do servidor web para disponibilizá-lo.

Árvore de pastas ideal

"Árvore de pastas" é a forma como as pastas de seu projeto estão organizadas. Há infinitas formas de se fazer isso.

Uma boa forma para organizar a pastas de projetos em Django é esta:



Esta árvore de arquivos está adequada para o sistema de arquivos do Linux, Unix e MacOSX, que juntos formam a maioria dos servidores web disponíveis em serviços de hospedagem. Entretanto é possível construir uma árvore de pastas idêntica no Windows, exceto pelos links simbólicos. Mas isso é possível de contornar para quem tem prática com servidores nesse sistema operacional.

Observe que dentro da pasta **"django"** há duas pastas:

pluggables

Você deve colocar nesta pasta as aplicações plugáveis extras do projeto, que veremos mais detalhadamente em um capítulo adiante.

Aplicações plugáveis são recursos extras que não são fornecidos pelo Django, mas por **terceiros**. Há centenas delas na Web, a grande maioria é gratuita e livre. Portanto, esta deve ser a pasta onde essas aplicações plugáveis serão instaladas.

Assim, elas podem ser usadas por vários projetos em um único servidor.

projects

Nesta pasta serão instaladas as pastas de projetos. Ou seja, até mesmo se você possuir mais de um site em seu servidor, dessa forma suas pastas continuam

organizadas.

Mas atenção: ao invés de colocar diretamente a pasta do projeto, crie uma pasta para o site e dentro desta pasta, coloque a do projeto, pois um site é mais do que o projeto. É muito provável que você vá querer documentar seu site, adicionar alguns arquivos de histórico e definições, back-ups e outros, portanto, o projeto não se limita aos arquivos-*fonte* em Django.

Agora, dentro da pasta do site do projeto, está a pasta do projeto em Django (**meu_blog**). Dentro dela, além do que já vimos no decorrer dos capítulos anteriores, há ali uma novidade:

deploy

Esta pasta é o lugar ideal para se guardar:

- os scripts de execução do projeto (*scripts* **FastCGI e WSGI**, por exemplo);
- e outros scripts e arquivos relacionados ao *deploy* no servidor, como por exemplo um arquivo com a lista de **cron jobs**, que são tarefas agendadas para servidores **Linux/Unix/MacOS X**.

No mesmo nível da pasta do projeto há outra pasta aqui:

dependencies

Esta pasta deve conter um arquivo chamado **README** onde você irá escrever as dependências de seu projeto.

Exemplos: se seu projeto trabalha somente na versão 1.0.2 do Django, escreva isso nesse arquivo. Se seu projeto depende de alguma ferramenta externa como o **ffmpeg**, **mimms** ou **Image Magick**, escreva também. Caso utilize funcionalidades do campo **ImageField**, você vai precisar da biblioteca **PIL**, portanto, escreva isso.

Sempre de forma explícita, informando a versão utilizada, o porquê e como deixar isso funcionando no projeto. Isso é extremamente importante para que não se perca no futuro.

Porquê usar o inglês nas pastas do servidor?

O inglês é a linguagem universal da atualidade. Ainda mais na informática.

Um dia qualquer, você e sua equipe podem precisar da ajuda de alguém que não conhece o seu idioma. Então essa pessoa irá fazer buscas no servidor por palavras comuns em língua inglesa.

Num mundo cada dia mais globalizado, isso se torna cada dia mais importante.

E agora, vamos ou não configurar um servidor?

• Sim, vamos... calma Alatazan, o processo de *deploy* é realmente abrangente, mas você vai perceber que na prática não é tão complexo quanto parece, pois a maioria dessas ferramentas pode ser instalada com grande facilidade ou muitas vezes já vêm instaladas nos serviços de hospedagem, portanto, atente-se aos conceitos, pois eles são importantes de se conhecer.

• Bom... mas como **resumir** todo esse aprendizado teórico de hoje? - os olhos de Alatazan estavam um pouco **vermelhos e ardiam**, pois é muito chato ficar ouvindo outra pessoa falar e não ver **exemplos na prática**.

• O melhor resumo para hoje são os diagramas, observe-os novamente, e não se limite a isso. Servidores são evoluídos e ajustados **ao longo do tempo**, à medida que o administrador e o site amadurecem. E para cada tipo de servidor ou serviço de hospedagem há tutoriais com explicações na web, aí basta fazer uma busca!

• Ok, mas amanhã vamos configurar um servidor Windows com Apache e MySQL né?

• Sim, vamos fazer isso amanhã. Agora é hora de **relaxar, respirar fundo** e procurar alguma coisa *light* pra fazer.

Capítulo 16: Preparando um servidor com Windows

Um índio passou em seu cavalo a galopes entre duas rochas. Havia poeira no ar, e também havia o cheiro da poeira, o Sol castigava a poeira, e a terra vermelha socada por séculos de pouca chuva e muito vento, fazia desenhos através do deserto, com muita poeira.

Os desenhos serpenteavam por aqui e ali, e pelas serpentes, ora passavam cavalos, ora passavam rios, ora passava uma criança chorando no colo de sua mãe, segurando firme seu iPod.

O índio de pele vermelha - não se sabe se era devido ao Sol, devido à terra avermelhada da região ou devido à sua timidez naquele momento - galopou novamente até uma lagoa, entregou peixes à boca de um golfinho, montou seu cavalo novamente e rumou a um celeiro, atravessando por uma janela larga e alta, e...

Opa! Voltando os olhos novamente àquele menino que não para de chorar, agora sentado com sua mãe, assistindo aos colegas terminarem a peça, morrendo de inveja porque não ia ganhar os aplausos do teatro e nem o novíssimo MelecaBoy verde, uma nova mania gosmenta da sessão de desenhos, e... enfim, isso nem interessa...

Ao final, **Cadu** correu até **Cartola**, tirou uma onda, curtiu seu momento de ator-mirim e voltou a rodopiar por ali.

Mas Alatazan ainda estava um pouco ligado no assunto do servidor para o dia seguinte...

Mas porquê construir um servidor no Windows?

A maior parte dos servidores web são **Linux**. Por alguns motivos, em grande parte dos casos os sistemas de padrão Unix, como Linux, Solaris e MacOS X, são mais adequados para servidores quando se trata de Django.

Mas o mundo é *multi*, e vamos falar agora de como se fazer as coisas no

Windows. Os procedimentos são semelhantes nos dois sistemas operacionais, e a parte mais importante - a configuração para ter seu projeto funcionando - é realmente muito parecida entre ambos, portanto seu aprendizado será útil em qualquer caso, mas agora vamos ao trabalho!

Você pode usar a mesma máquina que tem usado para estudar o Django, pois não se tratam de recursos conflitantes.

Esta é uma situação bastante normal para **intranets** e sistemas em **redes locais**.

Instalando o Apache

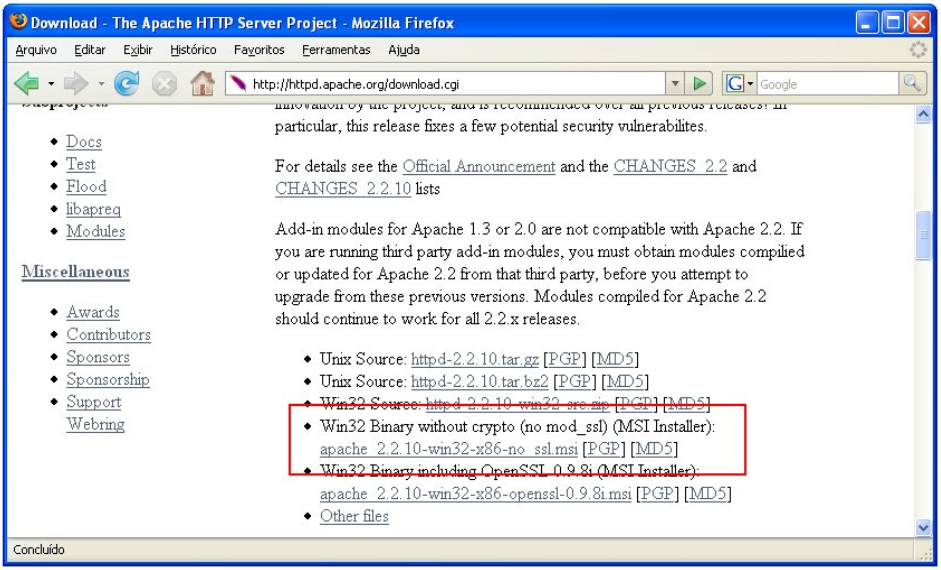
A primeira coisa a se fazer é instalar o **Apache** - de preferência na **versão 2.2**. Para isso vá ao site oficial e faça o download do arquivo de instalação:

```
http://httpd.apache.org/download.cgi
```

Localize agora um bloco que inicia com **"Apache HTTP Server 2.2.11 is the best available version"** ou algo bem semelhante a isso. Nesse bloco há uma lista das opções mais comuns de download do Apache. Escolha o **instalador para Windows, sem criptografia**, que em inglês está assim:

```
Win32 Binary without crypto (no mod_ssl) (MSI Installer):
```

Ou quase isso:



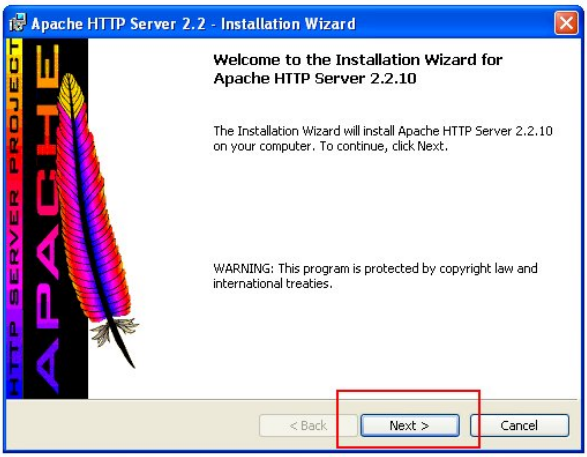
Clique no link para fazer o download.

Enquanto o download é feito, sua mente deve estar se perguntando **"porquê há**

um ar de indefinição aqui?".

Software é um produto **dinâmico**, e **software livre** é quase **orgânico** e evolui rapidamente. É provável que ao final deste livro a versão **mais recente** do Apache será outra, e até mesmo do Django. Ou seja, é impossível indicar um caminho exato para isso hoje, se você vai usá-lo amanhã. Portanto fique ligado e observe que os números da **versão** podem mudar rapidamente.

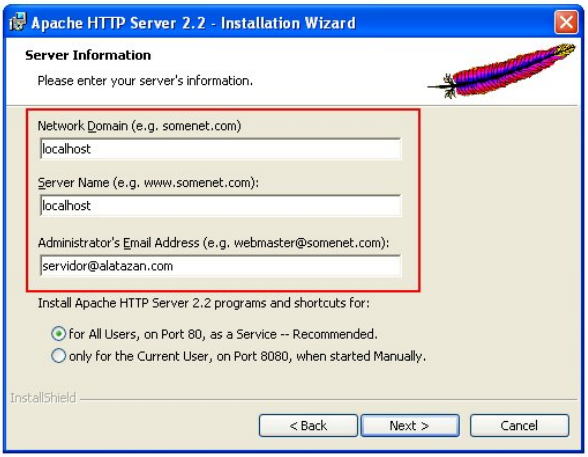
Ao final do download, clique duas vezes sobre o arquivo baixado, que será executado e a primeira janela será esta:



Siga clicando no botão **"Next"** e veja a janela seguinte:

Marque na primeira opção, indicando que aceita a licença do Apache, e siga

clicando em **"Next"** até chegar a esta outra janela:



Preencha os respectivos valores para os campos. Caso não saiba o que preencher ali, preencha da seguinte forma:

- Network Domain: **"localhost"**
- Server Name: **"localhost"**
- Administrator's E-mail Address: **"seu_email@qualquercoisa.com"**

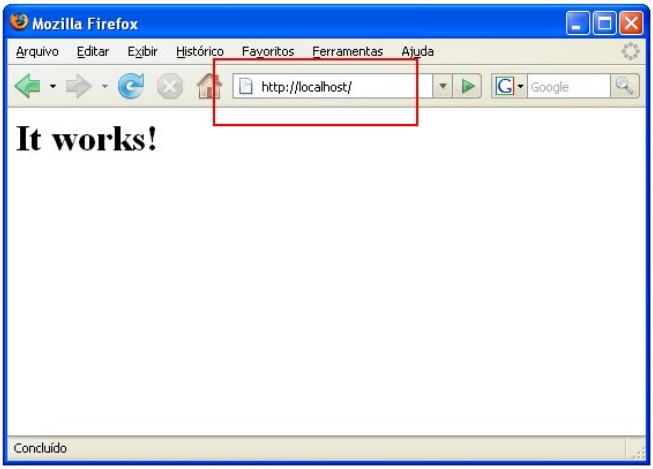
Modifique o campo **"Administrator's E-mail Address"** de acordo com seu próprio e-mail e siga adianta clicando em **"Next"** até finalizar.

Ao finalizar a instalação, verifique o **ícone do Apache na bandeja** do Windows, próximo ao relógio:

Agora vá ao navegador e carregue o seguinte endereço:

```
http://localhost/
```

Observe que desta vez não informamos a porta, pois estamos indicando que ela deve ser a porta **HTTP** padrão: **a 80**. Veja o resultado:



Viu como é fácil ter o Apache funcionando em sua máquina Windows?

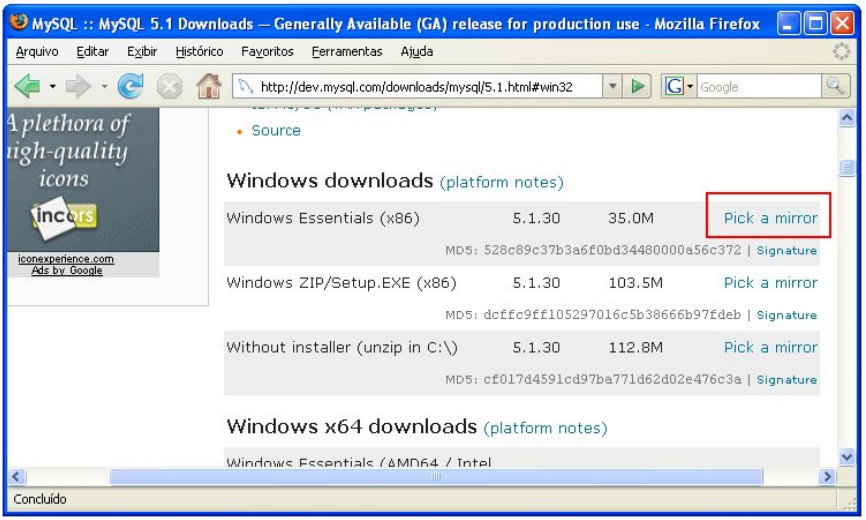
Instalando o MySQL

Agora vamos instalar o MySQL, o banco de dados que vamos usar neste servidor.

Para isso, vá ao site oficial de download do MySQL:

```
http://dev.mysql.com/downloads/mysql/5.1.html#win32
```

A versão mais indicada é a **versão 5.1**. Na opção **"Windows Essentials (x86)"**, clique no link **"Pick a mirror"**.

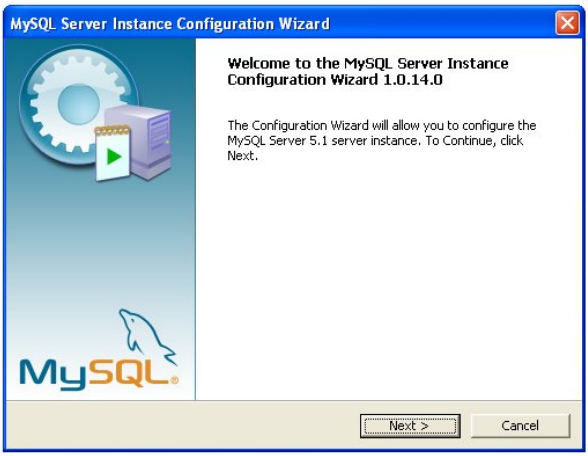


Será requisitado um cadastro gratuito, apenas para fins de *propaganda* mesmo. Faça-o (não há outra forma, *sorry*) e siga para selecionar um dos servidores **"mirrors"** para download.

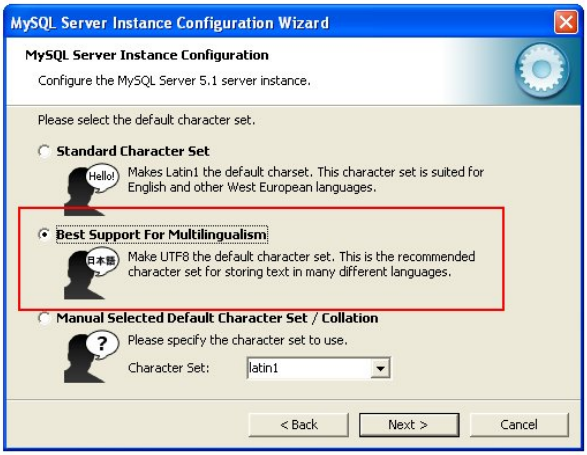
Após o download, clique duas vezes sobre o arquivo baixado e inicie a instalação do MySQL no Windows.

Siga clicando no botão **"Next"** até finalizar a instalação.

Ao final será exibida uma nova janela para configuração do MySQL (que por sinal é muito semelhante às janelas de instalação).

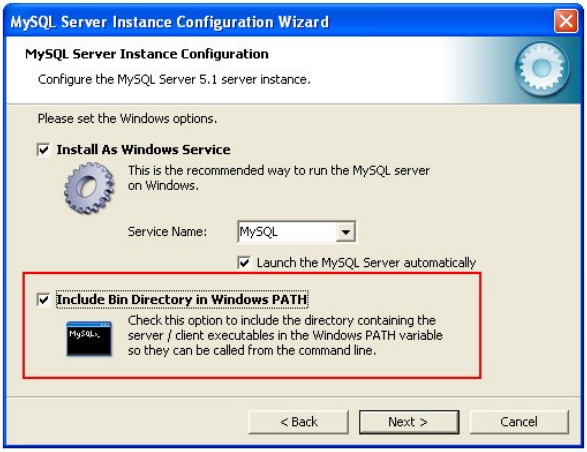


Prossiga clicando sobre **"Next"**, modificando as opções padrão somente se estiver certo de tal mudança. Até chegar a essa tela:



Escolha a segunda opção (**"Best Support for Multilingualism"**). Isso vai tornar a codificação de caracteres padrão em **"UTF-8"**, a que melhor se relaciona com o Django, e evitar alguma eventual chatice alguns dias à frente.

Prossiga clicando em **"Next"** até chegar a esta tela:



Marque a caixa **"Include Bin Directory to Windows PATH"**.

Prossiga clicando em **"Next"** até esta outra tela:

Determine uma senha difícil de ser quebrada, mas caso tenha dúvidas, simplesmente informe:

- Password: **"123456"**
- Confirm: **"123456"**

Porquê eu sugiro uma senha tão fácil de ser quebrada? Simples: **para que você não o faça!**

Prossiga clicando em **"Next"** até **executar** a configuração e concluí-la.

Pronto! Agora temos o MySQL rodando em seu servidor Windows!

Python e Django

Caso esta máquina não possua Python ou Django, faça a instalação seguindo os mesmos passos do capítulo 3 - **Baixando e Instalando o Django** - com a diferença que desta vez não é preciso instalar a biblioteca de acesso ao banco de dados SQLite (pySQLite), pois vamos usar o MySQL.

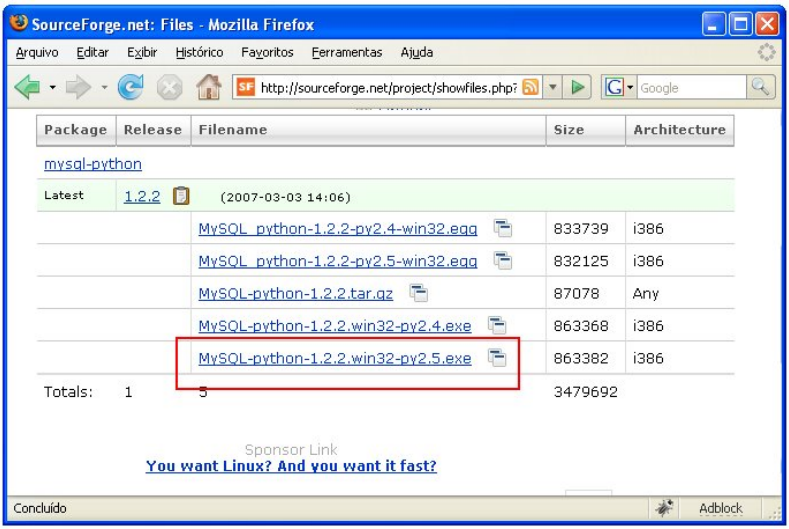
Biblioteca de acesso ao MySQL

Desta vez, a biblioteca de acesso ao banco de dados será a **"MySQL for Python"**, a biblioteca responsável por isso no Python.

Para fazer o download, vá à seguinte página da web:

```
http://sourceforge.net/projects/mysql-python
```

Faça o **download** do **mysql-python** na opção **"MySQL-python-1.2.2.win32-py2.5.exe"**. Da mesma forma que as instalações anteriores, a **versão atual pode variar** para cima (1.2.3, 1.3 ou 2.0 por exemplo):



Após o download, clique duas vezes sobre o arquivo baixado para instalar o **MySQL for Python** no Windows e siga a instalação clicando em **"Avançar"** até finalizar.

Instalando o mod_python

Como já foi dito no capítulo anterior, o **mod_python** ainda é a forma mais comum de instalar projetos em Django no Apache em servidores dedicados, mesmo

que a opção mais recomendada seja o **WSGI**.

Neste capítulo, vamos trabalhar com o **mod_python** e em um próximo capítulo será a vez do **WSGI**, isso porque é importante você conhecer os dois. Mas não se esqueça que é possível ter qualquer um deles (mod_python, WSGI ou FastCGI) instalado em Windows ou Linux.

Para fazer o download do **mod_python** para Windows, carregue a seguinte página em seu navegador:

```
http://httpd.apache.org/modules/python-download.cgi
```

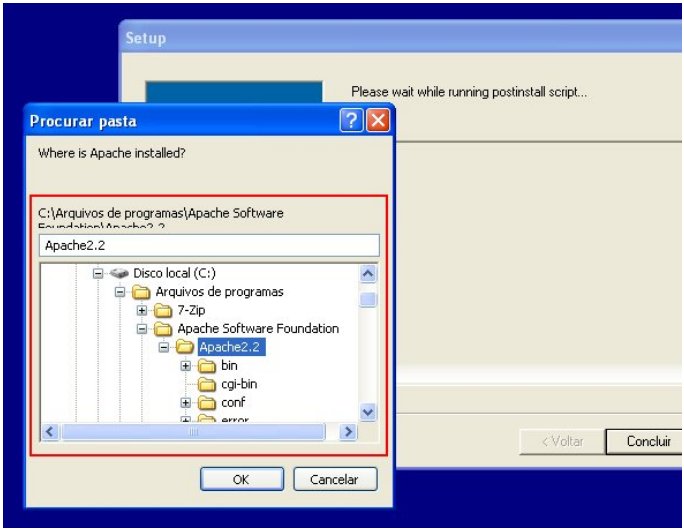
Localize o bloco na página que começa desta forma (lembrando que a versão pode variar):

```
Apache Mod_python 3.3.1 is now available
```

E abaixo dele, clique sobre o link **"Win32 Binaries"**. Uma lista nada amigável será exibida. Você deve localizar dentre as opções, o **mod_python** para a sua versão do Python (**2.5**) e a sua versão do Apache (**2.2**), que atualmente é esta:

```
mod_python-3.3.1.win32-py2.5-Apache2.2.exe
```

Após o final do download, clique duas vezes sobre o arquivo baixado e prossiga clicando em **"Avançar"** até esta janela ser exibida:



Escolha o **caminho de instalação** do Apache, que é este:

```
c:\Arquivos de programas\Apache Software Foundation\Apache2.2
```

Prossiga para finalizar a instalação.

Agora abra o seguinte arquivo para edição, usando o **Bloco de Notas**:

```
C:\Arquivos de programas\Apache Software
Foundation\Apache2.2\conf\httpd.conf
```

Localize uma linha que inicia com **"LoadModule"** e acrescente a seguinte linha abaixo dela:

```
LoadModule python_module modules/mod_python.so
```

Salve o arquivo. Feche o arquivo.

Na verdade você vai encontrar muitas linhas assim, mas não se preocupe. Se acrescentar a nova linha abaixo de qualquer uma delas, estará no lugar correto.

E agora você tem o **mod_python** instalado em seu servidor Windows.

Configurando seu projeto no servidor

Bom, agora estamos caminhando para o final.

Crie na raiz da unidade **"C:"** do servidor a seguinte sequência de pastas:

```
C:\var\django\projects\site_meu_blog
```

O caminho acima apenas segue a nossa proposta do capítulo 15, mas você pode trabalhar essa configuração como preferir.

Envie (usando seu meio escolhido) a pasta do projeto (**"meu_blog"**) para dentro da pasta criada acima.

Agora na pasta do projeto (**"C:\var\django\projects\site_meu_blog\meu_blog"**) abra o arquivo **"local_settings.py"** para edição e o modifique para ficar assim:

```
# Django settings for meu_blog project.
import os
PROJECT_ROOT_PATH = os.path.dirname(os.path.abspath(__file__))

LOCAL = False
DEBUG = False
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'meu_blog'
DATABASE_HOST = 'localhost'
DATABASE_USER = 'root'
```

```
DATABASE_PASSWORD = '123456'
```

Isso vai fazer seu projeto trabalhar como se deve no servidor: sem **modo de depuração** (DEBUG) e acessando ao banco de dados **MySQL**. Lembre-se de informar a setting **DATABASE_PASSWORD** com a senha correta que você informou ao **configurar** o MySQL.

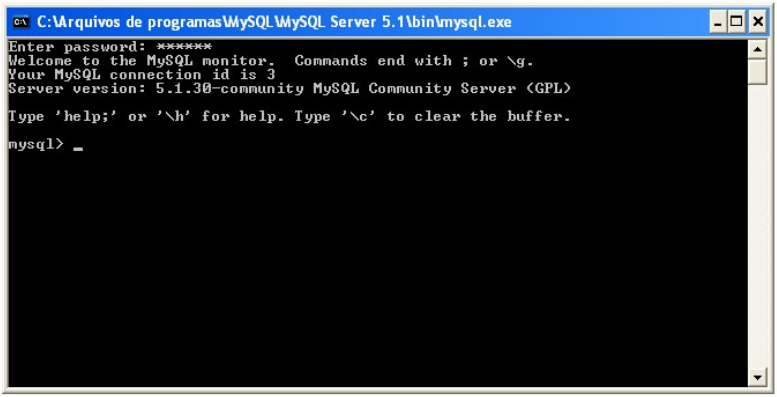
Criando o novo banco de dados no MySQL

Mas agora precisamos criar um novo banco de dados no MySQL para que seu projeto trabalhe. Como está escrito na setting **DATABASE_NAME**, o banco de dados deverá ser chamado **"meu_blog"**.

No menu **"Iniciar"** do Windows, escolha a opção **"Executar programa"** e digite o seguinte comando:

```
mysql -u root -p
```

Uma janela do MS-DOS será aberta para informar a senha do usuário **"root"** no MySQL. Informe a senha (a que você escolheu para o lugar de **"123456"**), pressione ENTER, e o **shell** do MySQL será aberto:



Agora digite a seguinte linha de comando:

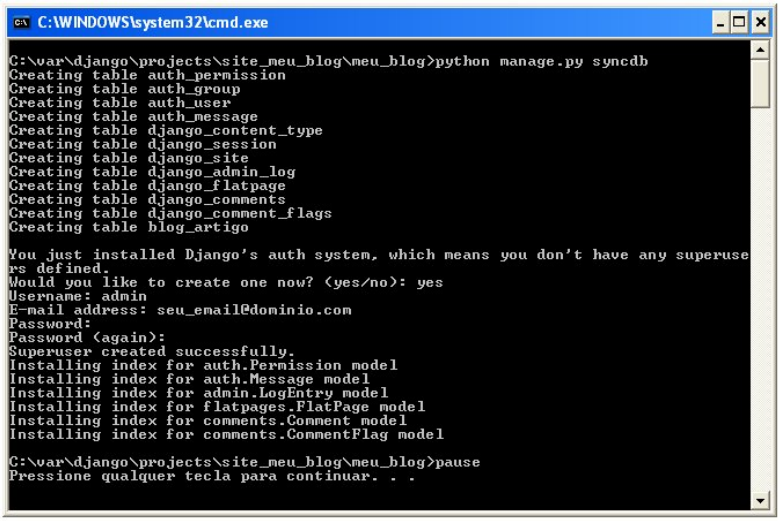
```
create database meu_blog;
```

Este será o resultado do comando:

```
mysql> create database meu_blog;
Query OK, 1 row affected (0.01 sec)

mysql>
```

Feito isso, seu banco de dados está criado.

Feche a janela do MS-DOS e vá à pasta do projeto. Clique duas vezes sobre o arquivo **"gerar_banco_de_dados.bat"** para gerar as tabelas para o novo banco de dados em MySQL. É agora que vamos saber se a nossa configuração está correta:



Informe os dados solicitados (Username, E-mail, Password e Password again) da mesma forma que fizemos no capítulo 4 (**Criando um blog maneiro**), com username **"admin"** e password **"1"**. Após finalizar a geração do banco de dados, feche a janela e...

... pronto! Mais um passo dado! Temos o projeto e o MySQL conversando naturalmente até aqui!

Configurando o Apache para seu projeto

Agora, para ter seu projeto funcionando no Apache, abra novamente o arquivo **"httpd.conf"** da pasta **"C:\Arquivos de programas\Apache Software Foundation\Apache2.2\conf\"** para edição e acrescente as linhas abaixo ao final do arquivo:

```
<Location "/">
SetHandler python-program
PythonPath "['c:/var/django/projects/site_meu_blog/meu_blog'] + sys.path"
PythonHandler django.core.handlers.modpython
SetEnv DJANGO_SETTINGS_MODULE settings
PythonDebug On
</Location>
```

O que fizemos ali? Bom, ao abrir uma tag **"Location"**, indicamos que o Apache deve seguir uma configuração especial para a sua URL raiz.

```
<Location "/">
```

A linha seguinte determina que para esta URL deve ser habilitado o suporte ao Python.

```
SetHandler python-program
```

Esta linha abaixo indica a **PYTHONPATH** que o Python deve obedecer na máquina virtual que será aberta para esta URL. No caso, atribuímos a pasta do projeto (**'c:/var/django/projects/site_meu_blog/meu_blog'**), acrescentada do que já estava antes na PYTHONPATH (**sys.path**), seguindo a sintaxe do próprio Python para isso.

```
PythonPath "['c:/var/django/projects/site_meu_blog/meu_blog'] + sys.path"
```

Na linha seguinte determinamos que o Apache deve utilizar o *handler* do Django para atender às requisições através do Python neste caso.

```
PythonHandler django.core.handlers.modpython
```

E aqui declaramos uma **variável de ambiente** que determina o módulo que contém as **settings** do projeto. Observe que se indicamos a pasta do projeto à PYTHONPATH, então todos os módulos (arquivos com extensão **".py"**) que estão na pasta do projeto podem agora ser informados com seu único nome, pois eles estão na PYTHONPATH sem nenhum pacote intermediário. Na prática, estamos passando o arquivo **"settings.py"** para o Django .

```
SetEnv DJANGO_SETTINGS_MODULE settings
```

Por fim, definimos o **estado de depuração** para a **máquina virtual** do Python. Como estamos ainda em processo de colocar o site no ar, é prudente manter essa configuração. Quando o site estiver em pleno funcionamento, é recomendável que esta linha seja removida.

```
PythonDebug On
```

Salve o arquivo.

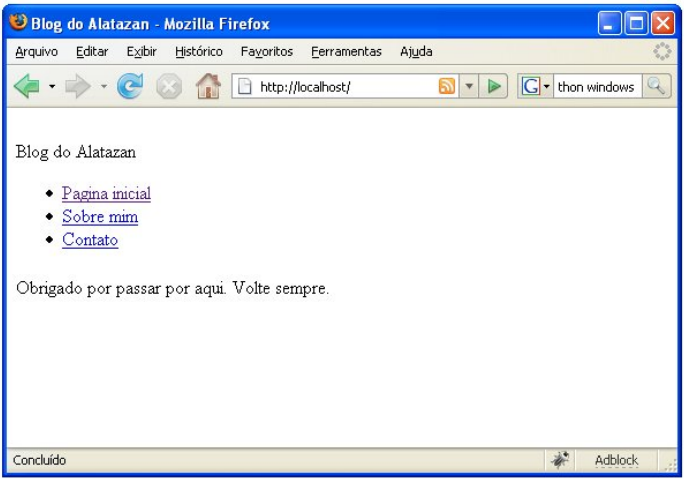
Reiniciando o Apache

Agora, observe o **ícone do Apache na bandeja** do Windows, próximo ao relógio, clique sobre ele e escolha a opção **"Restart"** para reiniciar o Apache e aplicar a mudança feita.

Vá ao navegador e carregue o endereço do servidor:

```
http://localhost/
```

A seguinte página será carregada:



Olha só! Temos o nosso site funcionando no servidor! Mas onde foi parar tudo aquilo que fizemos de **imagens e estilos CSS**?

Simples. Lembra-se que a URL **'^media/'** só é reconhecida pelo Django em ambiente de desenvolvimento? Pois bem, agora temos que configurar o Apache para reconhecê-la. Isto é uma tarefa que não tem nenhum intermédio do Django. É apenas um endereço do Apache que aponta para uma pasta no HD. Vamos lá?

Volte ao arquivo de configuração do Apache (**"httpd.conf"**) e acrescente o seguinte trecho de código ao final dele:

```
Alias /media C:/var/django/projects/site_meu_blog/meu_blog/media


<Directory "C:/var/django/projects/site_meu_blog/meu_blog/media">
    Order allow,deny
    Allow from all
</Directory>

<Location "/media">
    SetHandler None
</Location>
```

Um cuidado especial aqui: é muito fácil fugir da sintaxe neste arquivo de configuração, portanto fique atento aos espaços ou à falta deles... um bom exemplo é a parte "allow,deny" que não pode ter um espaço antes ou depois da vírgula

O bloco acima faz com que o Apache entenda que ao carregar a URL **"/media"** (e URLs "filhas"), a pasta **"C:/var/django/projects/site_meu_blog/meu_blog/media"** deve ser tomada como base, ou seja, a pasta de arquivos estáticos do nosso projeto.

A linha que faz esse trabalho é esta:

```
Alias /media C:/var/django/projects/site_meu_blog/meu_blog/media
```

Mas isso não é suficiente, pois é preciso dar permissão à pasta indicada, portanto, este bloco abaixo dá a permissão necessária:

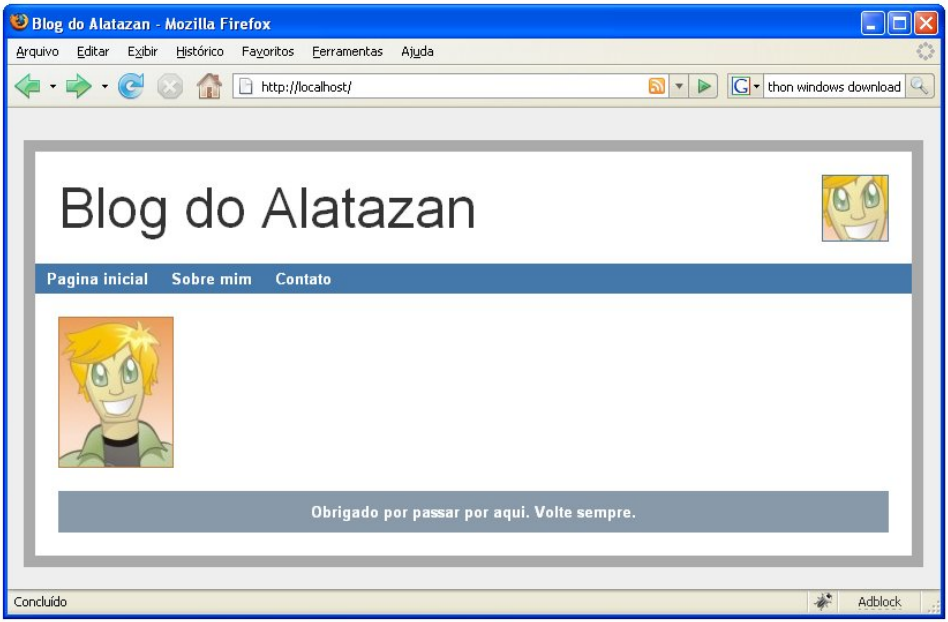
```
<Directory "C:/var/django/projects/site_meu_blog/meu_blog/media">
    Order allow,deny
    Allow from all
</Directory>
```

Por fim, este último bloco desabilita o Python para a URL **/media**, para que ela trabalhe somente com arquivos estáticos, sem qualquer interferência do Django.

```
<Location "/media">
    SetHandler None
</Location>
```

Salve o arquivo. **Reinicie** o Apache, usando o **ícone na bandeja** do Windows na opção **"Restart"** para aplicar a mudança que fizemos.

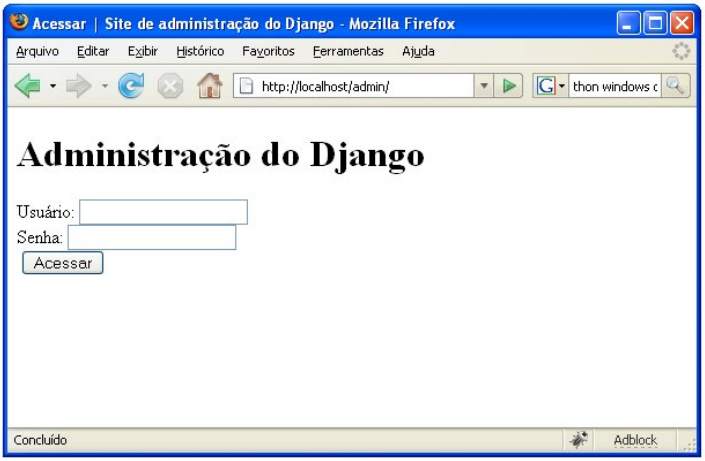
Voltando ao navegador, pressione **F5** e veja como ficou:



Satisfeito? Mas como pode notar, não temos nenhum **artigo** publicado, devido a ser outro banco de dados. Portanto, vamos ao **Admin** do site para criar os novos artigos! No navegador, carregue a seguinte URL:

```
http://localhost/admin/
```

Veja como ele será carregado:



Eta! Mas nós não havíamos resolvido o problema dos **arquivos estáticos**?

Sim, mas a URL que resolvemos foi a **"/media"** - a URL de arquivos estáticos do projeto. Porém, agora se trata do **Admin**, que é um caso especial, e possui seu próprio caminho para arquivos estáticos, com prefixo **"/admin_media"**, informado na setting **"ADMIN_MEDIA_PREFIX"**.

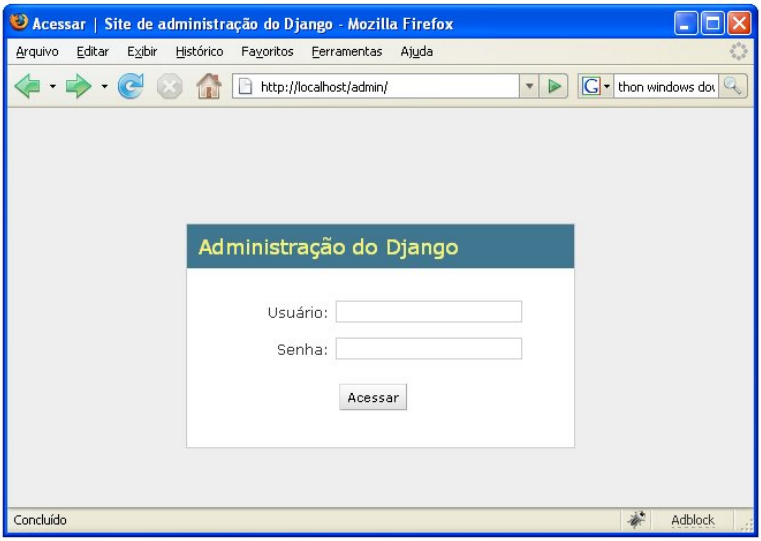
Portanto, volte ao arquivo de configuração do Apache (**"httpd.conf"**) e acrescente mais estas linhas ao final:

```
Alias /admin_media C:/python25/lib/site-packages/django/contrib/admin/media


<Directory "C:/python25/lib/site-packages/django/contrib/admin/media">
    Order allow,deny
    Allow from all
</Directory>


<Location "/admin_media">
    SetHandler None
</Location>
```

O que fizemos aí foi exatamente o mesmo que fizemos para a URL **/media**, mas desta vez trabalhamos a URL **/admin_media**, que é direcionada para a pasta de arquivos estáticos da *contrib* **Admin**.

Salve o arquivo. Feche o arquivo. **Reinicie** o Apache.

Agora atualize o navegador e veja como ficou:



Pronto! Servidor funcionando, podemos partir para o próximo desafio!

E aí, o quê mais?

- O quê mais? Meu amigo, estamos no meio da história, e nem terminamos o assunto do *deploy*! Amanhã vem a última bolacha do pacote.

Cartola encabulou com a energia de Alatazan. Ele não parecia exausto depois da extensa aula de hoje. Pelo contrário, Alatazan queria mais, queria saber quais outras formas de *deploy* existem, porque essa foi barbada.

Nena, que queria ir embora, puxou a cadeira que estava ao lado, coçou a bochecha, deu uma olhada no relógio e tomou sua vez:

- Alatazan, amanhã vai ser a minha vez de passar com você... quero te mostrar um servidor no **Linux** e outras coisas diferentes, vamos mudar isso aqui um pouco que esse ambiente de janelinhas de não é a minha praia não... - Nisso ela olhou irônica para o Cartola, que brigava com seu Mac pra pegar alguns arquivos do iPod. A resposta veio de bate-pronto.

- Hey, quem gosta de janelinha aqui é o Alatazan, não sou eu não hein, gosto de coisa com mais estilo.

Alatazan mexeu sua peruca desalinhado, fez de conta que não ouviu e tratou de voltar ao assunto:

- Bom, então hoje instalamos um monte de coisas:
  - Apache 2.2
  - MySQL 5.1
  - MySQL for Python 1.2.2
  - mod_python 3.3.1
- Olha, não se prenda às **versões** secundárias, apenas atenha-se a pegar a versão estável mais recente. Geralmente funciona melhor.
- E só completando, meu camarada... quando ela fala em versão "secundária", isso quer dizer que se aparecer um Apache 3.0 ou MySQL 6.0, é bom fazer uns testes antes, pois as versões primárias geralmente mudam muitas coisas...
- Ok ok... valeu, já peguei. Por fim..
  - Configuramos o acesso ao banco de dados no arquivo **"local_settings.py"** do servidor;
  - Criamos o banco de dados no MySQL;

- Geramos as tabelas do banco de dados pelo Django...

Nena tomou a palavra, já que ela tinha pressa e o Alatazan estava falando lentamente, como se estivesse saboreando um daqueles churrascos de **lagartos de Katara**...

- e configuramos o Apache, criando a **Location** para o projeto e os **Alias** pra URLs de arquivos estáticos! Aí foi só **reiniciar** o Apache e a mágica está feita! Vamos embora pessoal?

- Bacana, amanhã será a vez do Linux! Vai lá meu camarada, vou só pegar essa música aqui e já vou descendo...

Dito isso, Cartola voltou sua atenção para o computador e os dois saíram, se despedindo do **Cadú** e da priminha que insistia em arrancar seu nariz.

Capítulo 17: Preparando um servidor com Linux

Naquela noite ao chegar à sua casa, Nena abriu uma nova **máquina virtual com o Ubuntu Linux** em seu laptop. Não queria usar sua instalação oficial cheia de coisas já instaladas.

Alatazan por sua vez aproveitou a passagem de uma *nave-galé* de Katara pelo nosso sistema solar para dar uma passadinha por lá (atrás de Júpiter) e buscar algumas coisas que sua mãe mandou.

A definição mais comum da palavra "mãe" em nossa galáxia é de **"aquela que serve, protege, regula e manda pra você um monte de coisas inúteis que um dia com febre e assaduras você descobre que não eram tão inúteis assim"**.

Claro que a mãe de Alatazan não foge à definição, e nas caixas que mandou estão alguns pacotinhos com... por exemplo: uma caixinha de cuecas refrescantes, um spray do pó que refresca as cuecas por algum tempo, uma chave-polvo para desentalar o spray de vez em quando e uma alga-aérea para alimentar a chave-polvo.

Ao sair de Katara, Alatazan se sentiu aliviado por não ser mais obrigado a usar as tais cuecas a guardou as poucas que trouxe em algum lugar no fundo da nave. Mas agora, cansado de desenhar silhuetas desconfortáveis para o ventilador, ele estava com uma assadura na virilha, o que fazia com que andasse desajeitado, parecendo um **E.T.** e não sabia mais onde as cuecas estavam. Por isso, os pacotes de sua mãe foram bastante tranquilizadores.

Um servidor 100% livre

No capítulo anterior, trabalhamos com um conjunto bastante comum, com as ferramentas mais populares dentre as que estamos trabalhando: **Windows + MySQL + Apache + mod_python**.

Mas agora o nosso foco é trabalhar com alternativas.

Não que Windows trabalhe melhor com MySQL ou Linux trabalhe melhor com

PostgreSQL, não é bem por aí... o bom é conhecer sistemas operacionais diferentes, bancos de dados diferentes, servidores web diferentes e assim por diante.

Para colocar em prática o nosso servidor em Linux, não se esqueça de que é preciso ter o **controle total do sistema**, do contrário não será possível instalar os softwares que vamos instalar nem mesmo configurá-los. Portanto, a recomendação é que agora você use sua instalação do Ubuntu em **máquina local** ou em uma **máquina virtual**, como Alatazan o fez.

Neste capítulo vamos trabalhar no universo **Ubuntu/Debian**, que estão entre as distribuições mais populares do Linux.

Para fazer o download do Ubuntu, vá até esta página:

```
http://www.ubuntu.com/
```

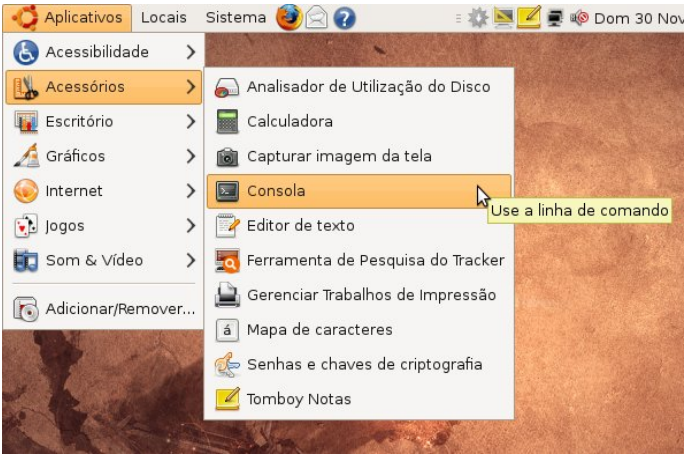
E para instalá-lo em uma máquina virtual, faça sua escolha favorita. Uma boa opção é o **VirtualBox**:

```
http://www.virtualbox.org/
```

Ambos são **livres e gratuitos**, aliás uma coisa bem comum entre todas as ferramentas que vamos tratar aqui.

O Ubuntu Linux possui uma interface gráfica extremamente fácil e amigável, entretanto, como muitos servidores Linux não possuem ambiente gráfico, vamos trabalhar constantemente em **Console** - modo texto semelhante ao **Prompt do MS-DOS**.

Neste capítulo, assumimos que você possui conhecimentos básicos em Linux suficientes para abrir o Console como pode ver abaixo e ter um editor de sua preferência, como gVim, vim, gedit, emacs, nano ou outro qualquer.



Instalando o Lighttpd

O Lighttpd é um dos servidores web mais leves e rápidos que existem, pois tem uma estrutura mais simples que o Apache e outros.

Para instalar o Lighttpd, abra uma janela do **Console** e digite o seguinte comando:

```
sudo apt-get install lighttpd
```

O comando **"sudo"** solicita sua própria senha para efeitos de segurança, mas ele armazena essa senha em memória por algum tempo, portanto você precisa digitar sua senha novamente somente se ficar muito tempo em espera.

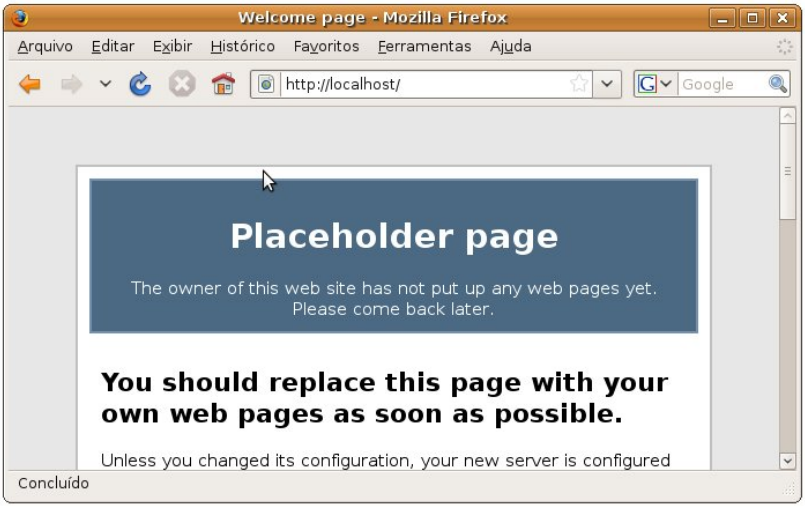
Confirme a instalação dos pacotes e espere pela conclusão da instalação.

Ao concluir, vá até seu navegador, e assumindo que esteja trabalhando em máquina local, carregue a seguinte URL:

```
http://localhost/
```

Caso esteja trabalhando em uma máquina remota, digite a URL para o caso.

De qualquer forma, veja como aparece:



Sim, é só isso mesmo para instalar o **Lighttpd**.

Instalando o PostgreSQL

Vamos agora instalar o PostgreSQL, um dos bancos de dados livres mais robustos e populares.

No **Console**, digite a seguinte linha de comando:

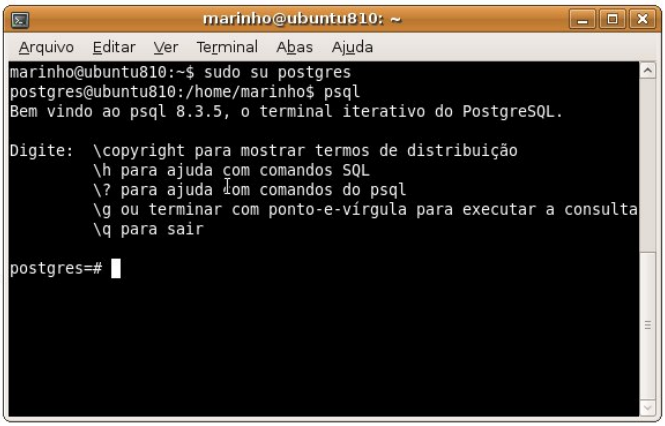
```
sudo apt-get install postgresql-8.3
```

Confirme a instalação dos pacotes e aguarde pela conclusão. Assim como aconteceu com o Lighttpd, ao concluir a instalação, o PostgreSQL já é iniciado.

Agora, precisamos definir uma senha para o usuário **"postgres"** para ter acesso ao banco de dados. Portanto, digite agora as seguintes linhas de comando:

```
sudo su postgres
psql
```

Será aberto um novo *shell* da seguinte forma:



Este é o *shell* cliente do PostreSQL, onde é possível realizar qualquer operação SQL de **definição de modelo de dados, consulta a dados, persistência e tarefas de manutenção**, enfim, muito poderoso.

Para alterar a senha do usuário **"postgres"** do PostgreSQL, digite a seguinte linha de comando:

```
alter user postgres with password '123456';
```

Não se esqueça de trocar a senha **'123456'** pela de sua preferência. O resultado na linha abaixo será este:

```
ALTER ROLE
```

Pronto. Agora digite o comando abaixo para sair do shell:

```
\q
```

E para sair da sessão do usuário **"postgres"** digite:

```
exit
```

Observe que estamos trabalhando aqui com dois usuários **"postgres"** diferentes: ora trabalhamos no usuário "postgres" **do Linux**, ora com o usuário de mesmo nome **no PostgreSQL**. Um possui vínculo indireto ao outro, mas se tratam de coisas diferentes. Ao modificar a senha do usuário **no PostgreSQL**, a senha do usuário **no Linux** permanece como estava. Mas isso é irrelevante agora, pois esse usuário só será usado para acesso a bancos de dados.

Agora, de volta à sessão do seu usuário, digite a seguinte linha comando:

```
psql -U postgres -W
```

Ao solicitar a senha, digite a senha informada no lugar de **'123456'** criada agora há pouco. Veja a mensagem de erro que será exibida:

```
psql: FATAL: autenticação do tipo Ident falhou para o usuário
"postgres"
```

Veja bem: quando acessamos o **psql** usando uma sessão do usuário **"postgres"** do Linux, a autenticação foi aceita sem nem mesmo requisitar senha, mas agora que usamos uma sessão do seu usuário, mesmo informando a senha correta a autenticação foi rejeitada. Porquê?

Porque existe uma configuração que determina isso. Vamos mudar essa configuração. Digite o seguinte comando:

```
sudo nano /etc/postgresql/8.3/main/pg_hba.conf
```

Você pode usar outro editor ao invés do **nano**, como o **vim** ou **gedit** por exemplo.

Localize agora as seguintes linhas, já próximo ao fim do arquivo:

```
# Database administrative login by UNIX sockets

local   all         postgres                          ident
sameuser
```

O que esta linha faz e determinar que o usuário **"postgres"** do PostgreSQL só pode ser autenticado caso esteja acessando do servidor **local**, com o usuário equivalente do sistema operacional (ou seja, o usuário **"postgres"** do Linux). Vamos portanto liberar a autenticação deste usuário por outros (como o seu próprio usuário por exemplo). Para isso, modifique as linhas para ficar assim:

```
# Database administrative login by UNIX sockets

local   all         postgres                          password
```

Salve o arquivo. Feche o arquivo.

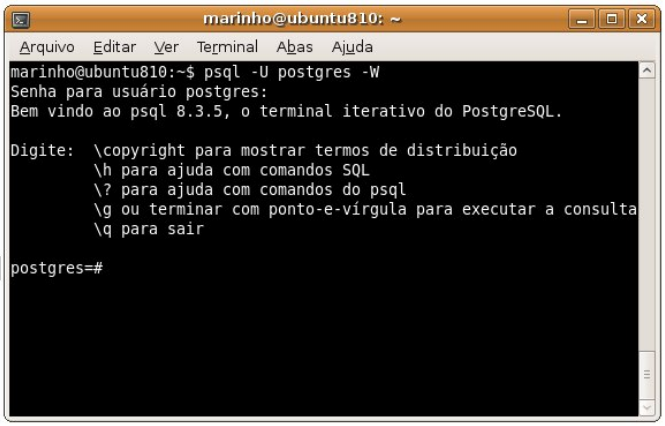
Agora de volta ao **shell**, reinicie o serviço do PostgreSQL com a seguinte linha de comando:

```
sudo /etc/init.d/postgresql-8.3 restart
```

Agora ao tentar novamente o seguinte comando, você terá sucesso:

```
psql -U postgres -W
```

Veja o resultado:



Pronto! Agora temos o PostgreSQL funcionando em nosso servidor, da forma apropriada ao que precisamos.

Python e Django

O **Ubuntu Linux** tem o **Python** como linguagem principal, portanto, você não precisa se preocupar em instalá-lo, pois ele já vem instalado. O mesmo vale para o **Debian**.

Para instalar o **Django** no Ubuntu Linux - **Intrepid Ibex**, versão **8.10** - basta digitar a seguinte linha de comando:

```
sudo apt-get install python-django
```

Após confirmar e concluir a instalação, seu servidor já terá o Django na versão 1.0 - no mínimo - devidamente instalada.

Já nas demais versões do Ubuntu Linux, isso não é recomendável, pois a maioria delas ainda tem esse pacote em uma versão antiga do Django (**0.96**). Portanto neste caso, para instalar o Django, faça assim:

Primeiro instale o **Subversion**, com a seguinte linha de comando:

```
sudo apt-get install subversion
```

Depois de confirmar e concluir a instalação do **Subversion**, faça o download do Django 1.0.2:

```
svn co
```

```
http://code.djangoproject.com/svn/django/tags/releases/1.0.2/
django-1.0.2
```

Isso vai baixar o **código-fonte** da última revisão da versão 1.0.2 do Django, usando o **Subversion** - um software de controle de versões.

Após concluir o download, entre na nova pasta **"django-1.0.2"** e instale o Django:

```
cd django-1.0.2
sudo python setup.py install
```

Após concluir a instalação do Django, para confirmar que está tudo ok, digite as seguintes linhas de comando para voltar à sua pasta **HOME** e acessar o **shell interativo** do Python:ip

```
cd ~
python
```

E agora, fora da pasta de instalação do Django e dentro do **shell interativo**, digite estas linhas de comando:

```
import django
django.get_version()
```

O resultado deverá ser este (ou outro muito semelhante a este, mas **sem mensagem de erro**):

```
'1.0.2 final'
```

Agora saia do **shell interativo**, pressionando **Control+D**.

Pronto! Agora temos o **Django** numa versão bacana funcionando em nosso servidor!

Biblioteca de acesso ao PostgreSQL

A biblioteca de acesso ao PostgreSQL para o Python se chama **"psycopg2"**. Para instalar o **"psycopg2"** no Ubuntu, basta digitar a seguinte linha de comando no **shell** do Linux:

```
sudo apt-get install python-psycopg2
```

Pronto! Após confirmar e concluir a instalação, já temos a biblioteca de acesso ao PostgreSQL funcionando no Python em nosso servidor!

Instalando o flup para trabalhar com FastCGI no Python

Agora, como exploramos o **mod_python** no capítulo anterior, desta vez vamos trabalhar com o **FastCGI** no Lighttpd. Então, será necessário instalar o **flup**, a

biblioteca do Python responsável pela compatibilidade com o **FastCGI**.

Para isso, digite a seguinte linha de comando:

```
sudo apt-get install python-flup
```

Agora o seu Python está preparado para trabalhar com FastCGI.

Configurando seu projeto no servidor

Da mesma forma que no Windows, algumas coisas precisam ser feitas no projeto para que ele trabalhe adequadamente no servidor. Algumas dessas coisas são semelhantes, outras não...

Para começar, crie a nossa pasta de coisas em Django no servidor, assim:

```
sudo mkdir /var/django
sudo chown SEU_USUARIO /var/django
sudo chgrp SEU_USUARIO /var/django
cd /var/django
```

Lembre-se de trocar o **"SEU_USUARIO"** nas duas linhas de comando pelo nome do seu usuário atual.

As linhas de comando vão:

1. Criar a pasta onde vamos colocar nossas coisas em Django, usando o superusuário do sistema operacional;
2. Assumir o seu próprio usuário como dono da nova pasta;
3. Assumir o seu próprio grupo como dono dela;
4. Acessar a pasta.

Agora cria a sequência de pastas:

```
mkdir projects
cd projects
mkdir site_meu_blog
```

Feito isso, envie a pasta do seu projeto (**"meu_blog"**) para esta pasta do servidor, em caminho completo:

```
/var/django/projects/site_meu_blog
```

Para isso, utilize uma das formas de envio de arquivos abordadas no **"capítulo 15: Infinitas formas de se fazer deploy"** (software de controle de versões, FTP ou Fabric).

Agora dentro da pasta do projeto no servidor (

"/var/django/projects/site_meu_blog/meu_blog/"), abra o arquivo **"local_settings.py"** para edição:

```
cd meu_blog
nano local_settings.py
```

Modifique o arquivo, para ficar desta maneira:

```
# Django settings for meu_blog project.
import os
PROJECT_ROOT_PATH = os.path.dirname(os.path.abspath(__file__))

LOCAL = False
DEBUG = False
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'meu_blog'
DATABASE_HOST = 'localhost'
DATABASE_USER = 'postgres'
DATABASE_PASSWORD = '123456'

FORCE_SCRIPT_NAME = ''
```

Da mesma forma que no capítulo anterior, **desabilitamos o modo de depuração**, mas desta vez preparamos o projeto para trabalhar com o **PostgreSQL**, ao invés do MySQL. Não vá se esquecer de ajustar a setting **"DATABASE_PASSWORD"** com a senha que escolheu para o lugar de **"123456"**.

O principal ponto a ser observado nessa modificação trata-se desta linha:

```
FORCE_SCRIPT_NAME = ''
```

Ela terá o papel de **ocultar** o nome interno do *script* do FastCGI, e caso não seja informada, o site muito provavelmente se comportará de forma instável em suas URLs.

Salve o arquivo. Feche o arquivo.

Criando o novo banco de dados no PostgreSQL

E como definimos que o nome do banco de dados do projeto deve ser

"meu_blog", então devemos criá-lo, certo? Pois é o que vamos fazer agora.

No **shell**, digite a seguinte linha de comando para acessar novamente o shell do PostgreSQL:

```
psql -U postgres -W
```

Informe a senha correta e digite a seguinte expressão SQL:

```
create database meu_blog encoding 'UTF-8';
```

Essa linha de comando vai **criar um novo banco de dados** e garantir que sua codificação de caracteres é a **'UTF-8'**, a mais compatível com o Django, por ele ser **Unicode**.

O resultado desse comando deve ser simplesmente:

```
CREATE DATABASE
```

Pronto. Banco de dados criado! Digite esta outra linha de comando para sair do shell do PostgreSQL:

```
\q
```

Agora, precisamos gerar as tabelas do banco de dados, lembra-se?

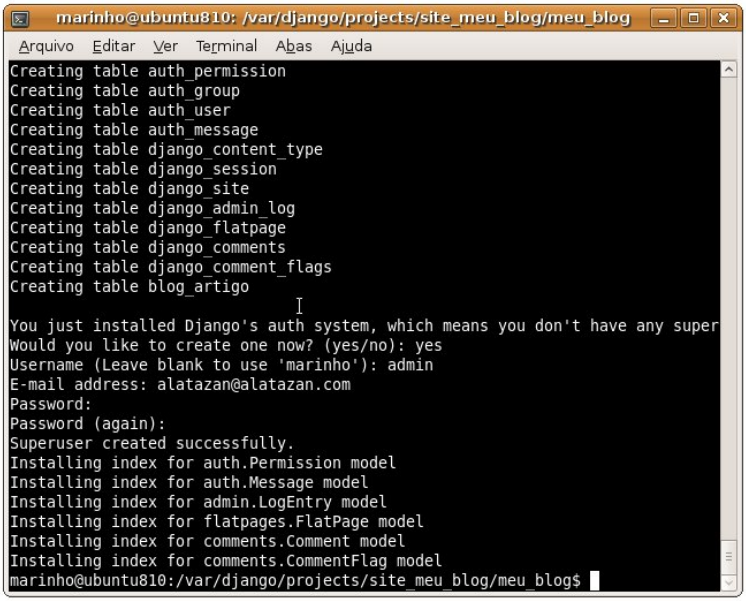
Só para nos certificar, digite a linha de comando abaixo para se posicionar na pasta do projeto:

```
cd /var/django/projects/site_meu_blog/meu_blog
```

Agora, digite esta outra linha de comando:

```
python manage.py syncdb
```

Depois de dadas as informações solicitadas (Username, E-mail, Password e Password again), o resultado deve ser este:



Não se preocupe em comparar tudo. Se não houve mensagem de erro é porque deu certo.

Bacana! Estamos caminhando bem! Nosso banco de dados foi criado e agora já possui as tabelas e outras entidades devidamente geradas pelo Django!

Configurando o Lighttpd para seu projeto

Agora precisamos trabalhar a configuração do Lighttpd para reconhecer o seu projeto, certo? Então vamos lá!

Digite as seguintes linhas de comando:

```
cd /etc/lighttpd/conf-available

ls
```

Veja o resultado:

```
05-auth.conf      10-proxy.conf          10-ssi.conf      10-
userdir.conf

10-cgi.conf       10-rrdtool.conf        10-ssl.conf      README

10-fastcgi.conf   10-simple-vhost.conf   10-status.conf
```

Esses arquivos são pré-configurados para determinadas utilidades. É uma dessas coisas feitas para ajudar quem está configurando o servidor. Mas nós vamos criar um novo, completamente vazio. Abra um novo arquivo **"10-meu_blog.conf"**

para edição, assim:

```
sudo nano 10-meu_blog.conf
```

Escreva estas linhas de código:

```
server.modules += ( "mod_fastcgi" )
server.modules += ( "mod_rewrite" )


fastcgi.server = (
    "/default.fcgi" => (
        "main" => (
            "socket" => "/tmp/lighttpd-default.sock",
            "check-local" => "disable",
            "bin-path" =>
"/var/django/projects/site_meu_blog/meu_blog/deploy/default.fcgi",
        )
    )
)


url.rewrite-once = (
    "^(/.*)$" => "/default.fcgi$1"
)
```

Resumindo, o que fizemos foi:

Carregar os modulos de FastCGI e Rewriting do Lighttpd:

```
server.modules += ( "mod_fastcgi" )
server.modules += ( "mod_rewrite" )
```

Determinar um caminho interno do servidor apontando para um script do projeto (alem de determinar o caminho para o arquivo de socket):

```
fastcgi.server = (
    "/default.fcgi" => (
        "main" => (
            "socket" => "/tmp/lighttpd-default.sock",
            "check-local" => "disable",
            "bin-path" =>
"/var/django/projects/site_meu_blog/meu_blog/deploy/default.fcgi"
,
```

```
        )
    )
)
```

E por ultimo, definimos uma URL (em expressão regular que quer dizer: **"todas a partir da barra"**) para apontar para esse script:

```
url.rewrite-once = (
    "^(/.*)$" => "/default.fcgi$1"
)
```

Salve o arquivo. Feche o arquivo.

Agora precisamos ativar essa configuração. Para isso, vamos criar um **link simbólico** para representar o arquivo **"10-meu_blog.conf"** na pasta que define as configurações ativadas.

```
sudo ln -s /etc/lighttpd/conf-available/10-meu_blog.conf ../conf-enabled/
```

Feito.

Acontece, que o script indicado não existe:

```
/var/django/projects/site_meu_blog/meu_blog/deploy/default.fcgi
```

Portanto, devemos criá-lo:

```
cd /var/django/projects/site_meu_blog/meu_blog/
mkdir deploy
cd deploy
nano default.fcgi
```

Lembre-se de trocar o **"nano"** pelo seu editor preferido, caso ele não for o **nano**.

Agora dentro do novo arquivo, digite as seguintes linhas de código:

```
#!/usr/bin/python
import sys, os


sys.path.insert(
    0, '/var/django/projects/site_meu_blog/meu_blog/'
)
sys.path.insert(
    0, '/var/django/projects/site_meu_blog/dependencies/'
)
```

```
sys.path.insert(
   0, '/var/django/pluggables/'
)


os.chdir('/var/django/projects/site_meu_blog/meu_blog/')
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'


from django.core.servers.fastcgi import runfastcgi
runfastcgi(method='threaded', daemonize='false')
```

Resumindo, o que fizemos nesse outro arquivo de *script* foi isso:

Possibilitamos que este arquivo seja executável, e dizemos ao Linux que ele deve ser interpretado pelo Python:

```
#!/usr/bin/python
```

Importamos os pacotes de **sistema** do Python e do **sistema operacional**:

```
import sys, os
```

Acrescentamos as pastas do projeto à PYTHONPATH:

```
sys.path.insert(
    0, '/var/django/projects/site_meu_blog/meu_blog/'
)
sys.path.insert(
    0, '/var/django/projects/site_meu_blog/dependencies/'
)
sys.path.insert(
    0, '/var/django/pluggables/'
)
```

Agora um pequeno *trick* necessário e a definição da variável de ambiente **"DJANGO_SETTINGS_MODULE"** apontando o módulo que contém as **settings** do projeto, que por sua vez será localizada na PYTHONPATH que acabamos de definir:

```
os.chdir('/var/django/projects/site_meu_blog/meu_blog/')
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
```

Por fim, executa a função para **"escutar" o FastCGI**:

```
from django.core.servers.fastcgi import runfastcgi
runfastcgi(method='threaded', daemonize='false')
```

Entendido? Salve o arquivo. Feche o arquivo.

Agora vamos transformar esse arquivo que criamos em executável:

```
chmod a+x default.fcgi
```

Feito isso, vamos **reiniciar** o Lighttpd para ver o efeito do que fizemos:

```
sudo /etc/init.d/lighttpd restart
```

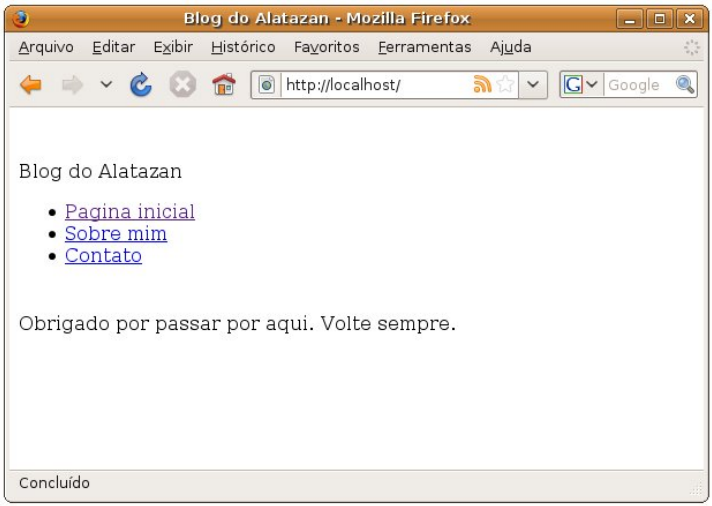
O resultado das linhas de comando deve ser este:

```
 * Stopping web server lighttpd                [ OK ]
 * Starting web server lighttpd                [ OK ]
```

Agora va ao navegador e carregue o endereço de seu site:

```
http://localhost/
```

Veja como ficou:



Ótimo! Mas como deve ter percebido, nosso site está **tosco**. Sem imagens, sem CSS. Da mesma forma como aconteceu no servidor Windows, isso é porque precisamos agora definir as URLs para arquivos estáticos **"/media"** e **"/admin_media"**.

Para fazer isso, edite novamente o arquivo de configurações do Lighttpd. Assim:

```
sudo nano /etc/lighttpd/conf-enabled/10-meu_blog.conf
```

E agora com o arquivo aberto para edição (usando seu editor predileto), modifique o arquivo para ficar assim:

```
server.modules += ( "mod_fastcgi" )
server.modules += ( "mod_rewrite" )


fastcgi.server = (
    "/default.fcgi" => (
        "main" => (
            "socket" => "/tmp/lighttpd-default.sock",
            "check-local" => "disable",
            "bin-path" =>
"/var/django/projects/site_meu_blog/blog/deploy/default.fcgi",
        )
    )
)


alias.url = (
    "/media" =>
 "/var/django/projects/site_meu_blog/meu_blog/media",
    "/admin_media" =>
 "/usr/lib/python2.5/site-packages/django/contrib/admin/media/"
)


url.rewrite-once = (
    "^(/media/.*)$" => "$1",
    "^(/admin_media/.*)$" => "$1",
    "^(/.*)$" => "/default.fcgi$1"
)
```

Observe que fizemos as seguintes modificações:

Primeiro, acrescentamos dois **"apelidos"**, dizendo ao Lighttpd que tais URLS (**"/media"** e **"/admin_media"**) devem apontar para as respectivas pastas no HD. Ou seja: são URLs simples para pastas de arquivos estáticos e devem fazer exatamente isso: retornar arquivos estáticos do HD no caminho definido.

Observe com atenção a pasta **"/usr/lib/python2.5/site-packages/django/contrib/admin/media/"** pois ela pode ser diferente em seu sistema operacional se por acaso ele usar a versão **2,4** do Python como padrão.

```
alias.url = (
```

```
    "/media" =>
 "/var/django/projects/site_meu_blog/meu_blog/media",
    "/admin_media" =>
"/usr/lib/python2.5/site-packages/django/contrib/admin/media/"
)
```

As demais linhas que acrescentamos criam URLs com suas respectivas expressões regulares para os "apelidos" de arquivos estáticos, veja:

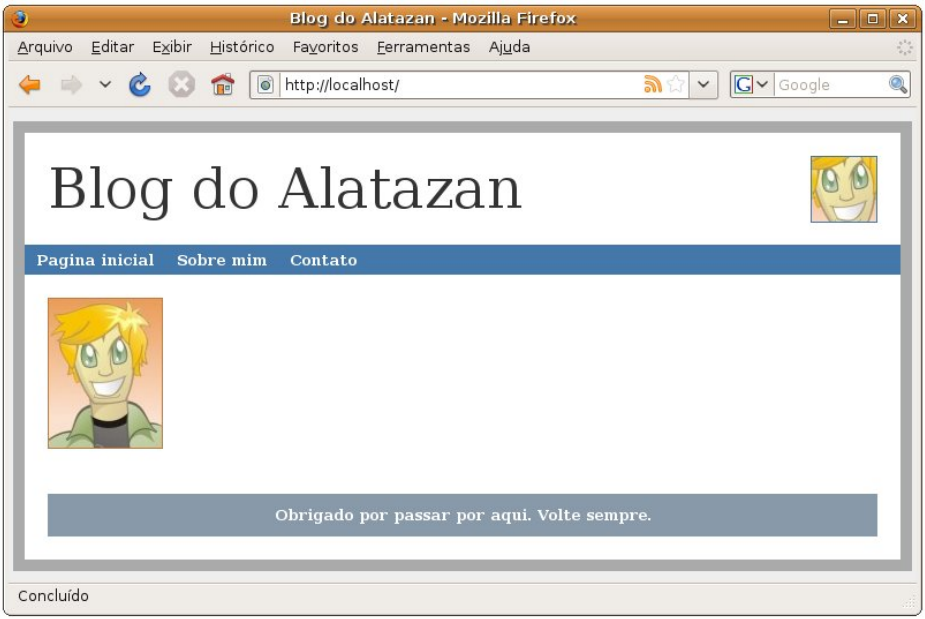
```
    "^(/media/.*)$" => "$1",
    "^(/admin_media/.*)$" => "$1",
```

Salve o arquivo. Feche o arquivo.

Reinicie o Lighttpd novamente:

```
sudo /etc/init.d/lighttpd restart
```

Volte ao navegador e... bingo!



Missão cumprida!

Vamos para a próxima etapa?

- Definitivamente! Isso me lembra um velho ditado em Katara: **fazeu, tá**

fazido!

Cartola trocou olhares com Nena, deu uma risadinha contida com o canto da boca... mas não quis corrigir o alienígena e quebrar sua empolgação. Mas foi Nena que continuou a conversa:

- Sim, Alatazan... veja:
  - Você instalou o Lighttpd;
  - e instalou também o PostgreSQL e **definiu uma senha** de acesso ao banco de dados;
  - depois você instalou o Django... em nosso caso usando o **apt-get** também;
  - instalou o **psycopg2** para PostgreSQL no Python;
  - instalou o **flup** para FastCGI no Python;
  - **enviou os arquivos** do projeto para a pasta correta no servidor;
  - configurou o **"local_settings.py"** para tirar o modo de depuração e configurar o acesso ao **banco de dados**;

Alatazan deu uma pescada e um arranque com a cabeça, se refez e abriu bem os olhos. Nena continuava:

- ... criou o banco de dados;
- **gerou as tabelas** no banco de dados;
- configurou o Lighttpd para acesso ao projeto;
- criou um **script de FastCGI** no seu projeto;
- e configurou as URLs para **arquivos estáticos**!

Depois dessa lista, emendou mais:

- Olha, você notou uma coisa interessante? Para quase tudo, usamos o **apt-get** para instalar as coisas. Ele faz isso: baixa o que precisamos, todas as suas dependências, configura tudo, move os arquivos nas pastas certas, faz tudo direitinho...

Estava claro que hoje Nena estava com mais tempo. Com bem mais tempo. Com muito tempo.

- Ahh, já sei! Vamos ver **WSGI e servidores compartilhados**?

E muito empolgadinha também.

- Err... vamos fazer isso amanhã... tenho minha aula de xadrez...

• Tudo bem rapaziada... mas enquanto vamos saindo, vou pedir uma coisa importante pra você fazer: volte ao assunto do servidor no Windows e **compare** com o servidor no Linux. Vai perceber que existem mais semelhanças do que parece!

Capítulo 18: WSGI e Servidores compartilhados

Houve uma época em que restavam poucos planetas e satélites próximos a Katara para serem habitados, e as pessoas perderam seu interesse por alguns deles em especial, talvez devido à falta de atmosfera, proximidade de Katara, falta de água ou qualquer outra desculpa que viesse à cabeça.

Isso não foi bom para os negócios de algumas empresas.

Foi por isso que a **Bangla Oportunidades Interplanetárias** lançou o slogan **"só para escolhidos a dedo"**. A parte do dedo não foi muito bem compreendida a princípio, mas ser um *escolhido* parecia ser bastante recompensador.

As sessões para escolha de novos clientes eram caracterizadas, dentre outras coisas, por avaliar eventos aleatoriamente e escolher aqueles que se esqueciam de parar de bater palmas ao final dos aplausos. Ou mesmo aqueles que batiam palmas sozinhos.

Atualmente a comunidade dos moradores do satélite **Katara-E** é o resultado disso, e é formada por uma população extremamente altruísta. Seus moradores compartilham suas casas, compartilham suas roupas, compartilham seus copos, suas mulheres, seus maridos, suas crianças e suas cuecas, especialmente as sujas.

Às vezes ocorrem algumas distrações como consequência disso, mas não interessa qual seja o desfecho, no final tudo é comemorado com uma boa cervejada e um enterro festivo.

WSGI e servidores compartilhados

Sim, esta é a última bolacha do pacote do assunto de *deploy*.

O **WSGI** é um recurso recente do Python, criado para superar os demais. Ele é simples, muitos o chamam de *"cola"*, uma coisa que vai ficar entre outras duas, somente para dar liga entre elas, e trabalha bem tanto de forma dedicada quanto compartilhada. Em Katara chamariam isso de "recheio de sanduíche de rodoviária".

Vamos agora voltar ao servidor Windows e tratar de substituir o **mod_python** pelo WSGI.

Baixando e instalando o mod_wsgi para Windows

O **mod_wsgi** é o módulo de suporte a **WSGI** para Apache.

Para fazer o download do **mod_wsgi** para Windows - sem a necessidade de compilar - vá a esta página da Web:

```
http://adal.chiriliuc.com/temp/mod_wsgi-1.0c1-rev_385-win32/
```

Escolha a opção mais adequada à sua versão.

No caso que temos trabalhado (**Apache 2.2** e **Python 2.5**), a versão a escolher é esta:

```
mod_wsgi_py25_apache22
```

Faça o download do arquivo **"mod_wsgi.so"**. Ele deve ser colocado na seguinte pasta do HD:

```
C:\Arquivos de programas\Apache Software
Foundation\Apache2.2\modules
```

Pronto. Instalado. É um processo rústico, mas simples.

Configurando o Apache para usar WSGI com o seu projeto

Agora, vamos modificar o Apache para que ele acesse seu projeto usando o **WSGI**.

Abra o arquivo **"httpd.conf"** da pasta **"C:\Arquivos de programas\Apache Software Foundation\Apache2.2\conf"** para edição e localize a seguinte linha:

```
LoadModule python_module modules/mod_python.so
```

Abaixo dela, acrescente mais esta:

```
LoadModule wsgi_module modules/mod_wsgi.so
```

Isto vai habilitar o **mod_wsgi** no seu Apache.

Agora localize este trecho de linhas:

```
<Location "/">
SetHandler python-program
PythonPath "['c:/var/django/projects/site_meu_blog/meu_blog'] + sys.path"
PythonHandler django.core.handlers.modpython
SetEnv DJANGO_SETTINGS_MODULE settings
PythonDebug On
</Location>
```

Remova-o. Ou se preferir, copie para um bloco de notas para consultar depois se precisar.

Agora escreva o seguinte código no mesmo lugar de onde removeu o anterior:

```
WSGIScriptAlias / "c:/var/django/projects/site_meu_bl
og/meu_blog/deploy/default.wsgi"


<Directory "c:/var/django/projects/site_meu_blog/meu_blog/deploy/">
    Order deny,allow
    Allow from all
</Directory>
```

Como pode ver, fizemos uma referência para dizer ao Apache que as URLs iniciadas com uma barra (ou seja: **todas, exceto às de arquivos estáticos**) devem ser direcionadas para o script WSGI no arquivo **"default.wsgi"** da pasta **"deploy"** do projeto. Veja:

```
WSGIScriptAlias / "c:/var/django/projects/site_meu_bl
og/meu_blog/deploy/default.wsgi"
```

Já no segundo bloco, nós definimos a permissão para que o Apache acesse a pasta deste script:

```
<Directory "c:/var/django/projects/site_meu_blog/meu_blog/deploy/">
    Order deny,allow
    Allow from all
</Directory>
```

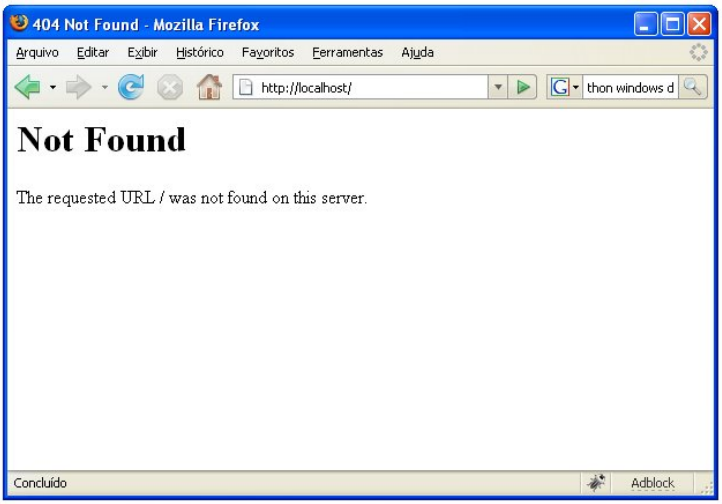
Salve o arquivo. Feche o arquivo.

Reinicie o Apache pelo ícone na bandeja do Windows, próximo ao relógio.

Vá ao navegador e carregue a URL do servidor:

```
http://localhost/
```

Mas veja o que acontece:



É que... Acontece que este arquivo de *script* ainda não existe. Portanto, vamos criá-lo! Se por acaso a mensagem de erro no navegador for algo parecido com 'Forbiden...' continue abaixo, pois esta normalmente é a mensagem exibida quando a pasta de *deploy* ainda não existe.

Script para o projeto trabalhar com WSGI

Abra a pasta do projeto no servidor:

```
c:\var\django\projects\site_meu_blog\meu_blog
```

Agora crie a seguinte pasta - se ela ainda não existir:

```
deploy
```

E agora dentro da pasta **"deploy"**, crie um arquivo chamado **"default.wsgi"**, com o seguinte código dentro:

```
import os, sys


PROJECT_ROOT_PATH = os.path.dirname(
    os.path.dirname(os.path.abspath(__file__))
)
sys.path.insert(0, PROJECT_ROOT_PATH)
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'


import django.core.handlers.wsgi
```

```
application = django.core.handlers.wsgi.WSGIHandler()
```

Agora observe o que fizemos:

Na primeira linha simplesmente importamos os pacotes de **sistema operacional** e de **sistema** do Python.

```
import os, sys
```

No bloco seguinte, nós encontramos a **pasta do projeto** (uma acima da pasta do script), inserimos seu caminho **no início da PYTHONPATH** e definimos o módulo de **settings** para o Django:

```
PROJECT_ROOT_PATH = os.path.dirname(
    os.path.dirname(os.path.abspath(__file__))
)
sys.path.insert(0, PROJECT_ROOT_PATH)
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
```

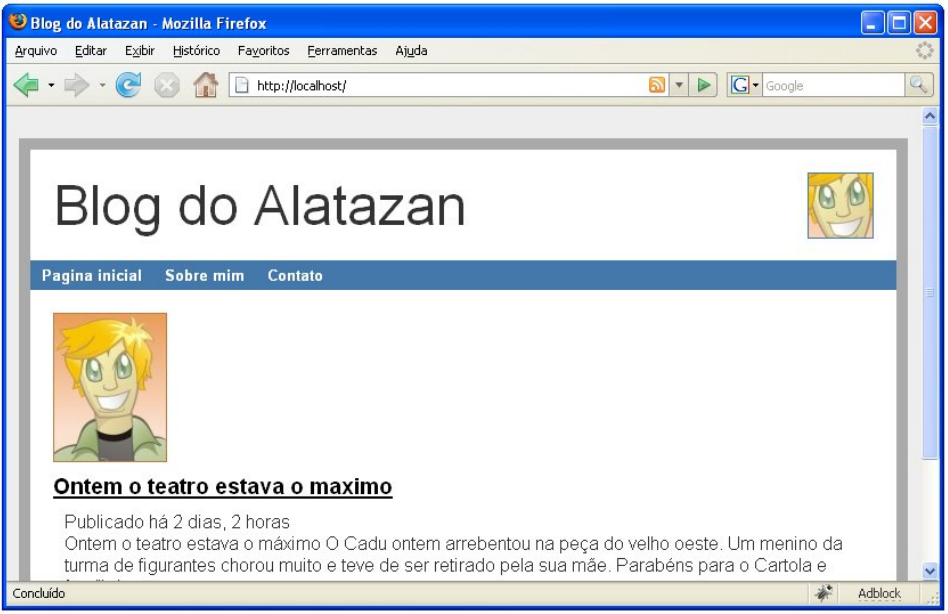
Por fim, importamos o *handler* do Django para trabalhar com **WSGI** e declaramos a variável **"application"** atribuída de uma instância do *handler*, veja:

```
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

O WSGI trabalha sempre procurando pela variável **"application"** no script em questão. Portanto, não mude este nome, ele é **fundamental**.

Salve o arquivo. Feche o arquivo.

Reinicie o Apache pelo ícone na bandeja do Windows, volte ao **navegador**, atualize com **F5** e veja como ficou:



É isso aí! Como pode ver, agora temos o servidor trabalhando em **WSGI**!

Ajustando o servidor para trabalhar compartilhado

Um servidor compartilhado é aquele que permite **vários sites** usando os mesmos recursos. É uma alternativa de baixo custo e bastante acessível, onde a quantidade de sites ativos pode ser até de 500 sites, muitas vezes utilizando tecnologias variadas como PHP, Python, Ruby e Perl numa só máquina.

Muitas vezes isso não é uma boa escolha, mas claro que tudo depende de como as limitações técnicas são exploradas.

O funcionamento de um servidor compartilhado em geral segue o seguinte raciocínio:

1. Cada site possui um **arquivo de configuração** e uma **pasta de documentos**. Respectivamente semelhantes ao **"httpd.conf"** e à pasta "C:\Arquivos de programas\Apache Software Foundation\Apache2.2\htdocs".

2. O arquivo de configuração é acessível somente ao administrador do servidor. Portanto, ele deve suportar que alguns ajustes sejam feitos por você através de um arquivo chamado **".htaccess"** na pasta de documentos do seu site.

3. O arquivo **".htaccess"** por sua vez declara algumas regras para liberar

alguns caminhos da pasta de documentos, e direcionar o restante para o Django. Para isso, ele precisa de um módulo do Apache: p **"mod_rewrite"**.

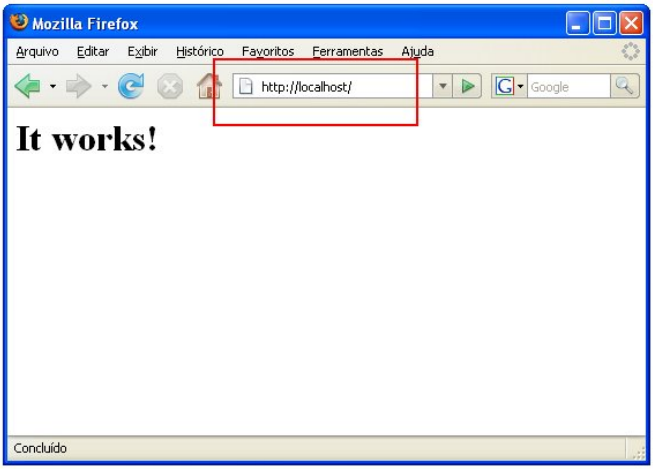
Pois bem. Nós vamos agora ajustar o Apache para trabalhar com o mesmo comportamento de um site de servidor compartilhado.

Então, para começar, abra o arquivo **"httpd.conf"** da pasta **"C:\Arquivos de programas\Apache Software Foundation\Apache2.2\conf"** para edição. Localize **e remova** o seguinte bloco, já no final do arquivo:

```
WSGIScriptAlias / "c:/var/django/projects/site_meu_blog/meu_b
log/deploy/default.wsgi"


<Directory "c:/var/django/projects/site_meu_blog/meu_blo
g/deploy/">
    Order deny,allow
    Allow from all
</Directory>


Alias /media "C:/var/django/projects/site_meu_blog/meu_blo
g/media"


<Directory "C:/var/django/projects/site_meu_blog/meu_blog/media">
    Order allow,deny
    Allow from all
</Directory>


<Location "/media">
    SetHandler None
</Location>
```

Salve o arquivo. **Reinicie** o Apache.

Veja o resultado:



Voltamos à estaca zero? Nem tanto. Mas agora localize a seguinte linha no arquivo:

```
#LoadModule rewrite_module modules/mod_rewrite.so
```

E remova o caracter de comentário (#), para ficar assim:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Isso vai ativar o **"mod_rewrite"** no Apache.

Agora acrescente esse trecho ao final do arquivo:

```
AddHandler wsgi-script .wsgi


<Directory "C:/Arquivos de programas/Apache Softw
are Foundation/Apache2.2/htdocs">
    AllowOverride FileInfo
    Options ExecCGI MultiViews FollowSymLinks
    MultiviewsMatch Handlers


    Order allow,deny
    Allow from all
</Directory>
```

Veja só o que fizemos:

Aqui nós vinculamos uma **extensão de arquivo** ao formato de script de WSGI:

```
AddHandler wsgi-script .wsgi
```

E aqui, modificamos a configuração da **pasta de documentos** para suportar a configuração do site usando **".htaccess"** e scripts **WSGI**:

```
<Directory "C:/Arquivos de programas/Apache So
ftware Foundation/Apache2.2/htdocs">
    AllowOverride FileInfo
    Options ExecCGI MultiViews FollowSymLinks
    MultiviewsMatch Handlers

    Order allow,deny
    Allow from all
</Directory>
```

Salve o arquivo. Feche o arquivo. **Reinicie** o Apache.

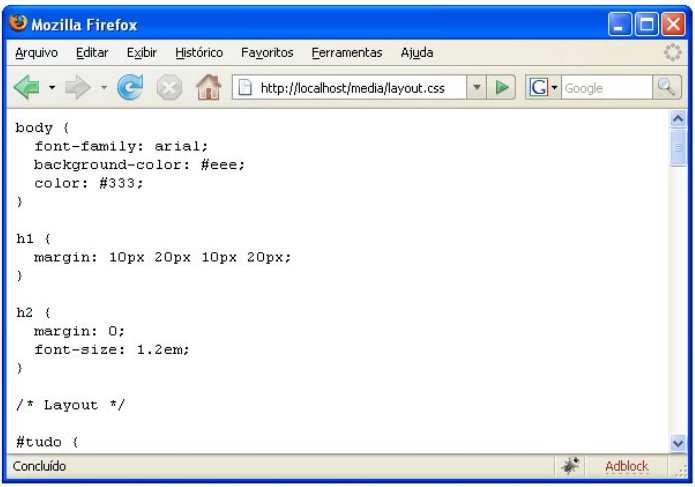
Trabalhando na sua pasta de documentos

O próximo passo agora é copiar a pasta de **arquivos estáticos** do seu projeto para a **pasta de documentos**. Isso é necessário pois o **Windows** não tem suporte a **links simbólicos**, um recurso do Linux, MacOS X e outros sistemas operacionais.

Copie a pasta **"media"** do projeto e vá ao navegador, carregue a URL abaixo e veja se o Apache a encontra:

```
http://localhost/media/layout.css
```

Ele deve ser carregado assim:



Isso indica que a pasta foi carregada corretamente e que o Apache a encontra da forma esperada.

Feito isso, ainda na pasta de documentos (**"C:\Arquivos de programas\Apache Software Foundation\Apache2.2\htdocs"**), crie o arquivo **".htaccess"** com o seguinte código dentro:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ /site.wsgi/$1 [QSA,PT,L]
```

Vamos detalhar o que essas linhas significam? Veja:

Ativamos o **rewriting** da pasta de documentos. A partir desse ponto as URLs são direcionadas de acordo com as regras abaixo desta linha:

```
RewriteEngine On
```

Dizemos que se o caminho da URL requisitada existir na pasta de documentos, ela terá prioridade sobre o Django. É esta linha que dá vida à pasta **"media"** trabalhando com arquivos estáticos.

```
RewriteCond %{REQUEST_FILENAME} !-f
```

Dizemos que todas as demais URLs (que não existirem enquanto arquivos da pasta) serão repassadas ao nosso projeto em Django. Representado por um *script* da pasta de documentos, chamado **"site.wsgi"**.

```
RewriteRule ^(.*)$ /site.wsgi/$1 [QSA,PT,L]
```

Salve o arquivo. Feche o arquivo.

Como pode notar, o arquivo de *script* **"site.wsgi"** de fato não existe. Pois então, ainda na **pasta de documentos**, crie um novo arquivo chamado **"site.wsgi"** com o seguinte código dentro:

```
import os, sys


PROJECT_ROOT_PATH = 'C:/var/django/projects/site_meu_blog/meu_blog'
sys.path.insert(0, PROJECT_ROOT_PATH)
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'


import django.core.handlers.wsgi


application = django.core.handlers.wsgi.WSGIHandler()
```

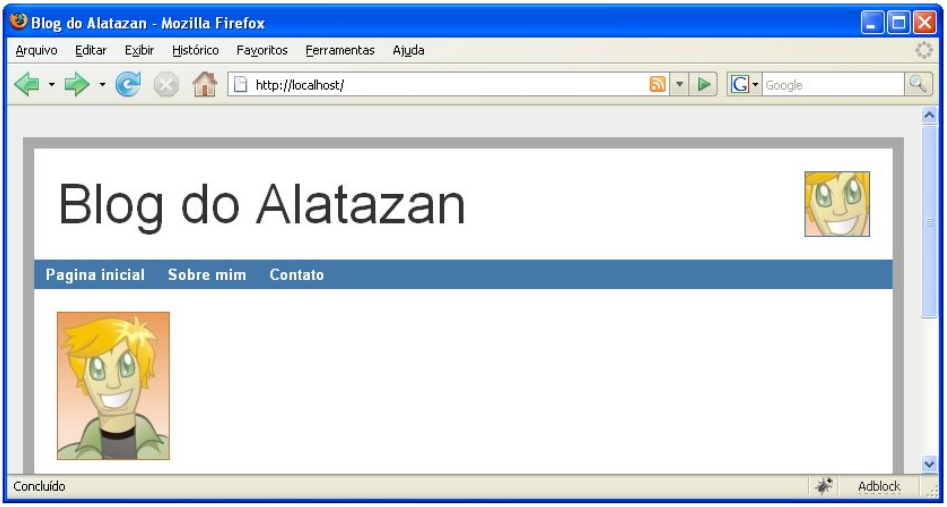
Este script é muito semelhante ao que criamos no início do capítulo, mas o caminho do projeto (**"C:/var/django/projects/site_meu_blog/meu_blog"**) é

apontado manualmente, sem uso de recursos do Python. Isso porque agora estamos fora da pasta do projeto e não temos referência relativa a ela.

Salve o arquivo. Feche o arquivo. Volte ao navegador e carregue a URL do site:

```
http://localhost/
```

E veja que tudo voltou ao normal:



Pronto! Temos agora o site funcionando novamente, porém desta vez usando **WSGI com rewriting**, um recurso fundamental para sites compartilhados.

Agora a ordem é: evoluir!

Após essas aulas sobre *deploy*, Alatazan decidiu por contratar um servidor compartilhado, com Linux, Apache e WSGI. Foi fácil colocar seu site no ar, pois como o servidor já estava devidamente configurado, seu trabalho se resumiu em criar um **link simbólico** para a pasta **"media"**, o arquivo **".htaccess"** e o script de **WSGI** apontando para seu projeto.

Ainda assim, manteve a configuração no Windows para testes eventuais.

Voltando para casa, um bêbado notou seus grandes olhos e tom peculiar de pele. Num ataque de ecologismo, cambaleante e vendo dois ou três Alatazans de uma só vez, o bêbado se pôs a gritar:

- Eu vi uma zebra! Eu vi uma zebra!

O que ele ainda não havia entendido era o que a cor da pele e os olhos de Alatazan tinham a ver com a **zebra**. Talvez fosse seu subconsciente - também

muito tonto - se lembrando das notícias daquela manhã.

As notícias que vira na TV eram sobre sinais estranhos e inexplicáveis, deixados no meio de uma plantação de cana. Lá havia também uma toalha para limpar o suor e uma lâmina de um metal desconhecido. As expressões "apocalipse", "sinais dos tempos" e "prejuízo nesta safra" eram as mais comuns no vocabulário do noticiário, mas enfim, isso tabém não lembrava uma zebra. Exceto por aquela zebra comendo cana no canto superior direito da tela da TV.

Alatazan deu de ombros e continuou seu caminho, lembrando do que aprendeu hoje, que resumidamente foi:

- Configurar um servidor Apache com WSGI, usando **mod_wsgi**;
- Criar script de WSGI na pasta **"deploy"** do projeto;
- Habilitar o **mod_rewrite** no Apache;
- Ativar recursos e sobrecarga na **pasta de documentos** do Apache, para suportar scripts WSGI e o arquivo **".htaccess"**;
- Copiar a pasta **"media"** do projeto para a **pasta de documentos**;
- Criar script de WSGI na **pasta de documentos** para acessar o projeto.

E foi só isso mesmo! No outro dia, seria a vez de usar *signals* e **URLs descomplicadas**.